Simple and Light Interfaces for C and C++ users

SLLIB – Script-Like C-language library

Basic User Reference Guide

Version 1.4.2[not finished] 2013-05-19

Credits

Software Development:

Chisato Yamauchi

Quality Assurance:

SEC Co., LTD.

Manual Document:

Chisato Yamauchi, *Sachimi Fujishima* and *SEC Co., LTD.*

Manual Translation:

KOYOSHOUJI CO.,LTD., *Space Engineering Development Co.,LTD.* and *Sakura Academia Corporation*

Special Thanks:

Daisuke Ishihara, *Hajime Baba*, *Iku Shinohara*, *Keiichi Matsuzaki*, *Michitaro Koike*, *Sergio Pascual* and *Yukio Yamamoto*

Web page: `http://www.ir.isas.jaxa.jp/~cyamauch/sli/`

Contents

1 Introduction

1.1 What is SLLIB?

SLLIB (pronounced es el lib; Script-Like C-language library) is the library that **adds the practical APIs to the C language that enable users to handle "streams," "character strings," "multidimensional arrays" as though they were doing so in any of the various scripting languages (Perl, PHP, Python, IDL, etc)**. SLLIB minimizes the weaknesses of the C language that become manifest during the processes frequently occurring on a routine basis, enabling **reduced efforts of coding and debugging and improved development efficiencies**.

For example, take a look at the following code:

```
#include <sli/tarray_tstring.h>
using namespace sli;

int main()
{
    tarray_tstring arr;
    arr[0] = "foo";             /* Assign "foo" to arr[0] */
    arr[1] = "bar";             /* Assign "bar" to arr[1] */
```

This example shows how SLLIB is used to write a code that assigns the character strings `"foo"` and `"bar"` each to the character string array. Where is this code different from codes that you use in the C language? The difference is that this code contains no descriptions about securing arrays of pointer and string buffers. The reason why it contains no such descriptions is that **required memory areas are automatically secured and managed on the library side**, so that the user's code do not need to have such descriptions written into it (Obviously, the release of memory areas is also automatically performed).

The following code is an example of using a stream through a network and a regular expression:

```
#include <sli/digeststreamio.h>
#include <sli/tstring.h>
using namespace sli;

int main()
{
  tstring line;
  digeststreamio f_in;
  f_in.open("r", "http://www.foo.bar/data/foo.txt.gz");   /* Open the file */
  while ( (line=f_in.getline()) != NULL ) {           /* Read lines one by one */
    line.chomp();                                     /* Delete the newline char */
    if ( 0 <= line.regmatch("^[a-zA-Z]",NULL) ) {     /* Try reg. expression matching */
      printf("%s\n",line.cstr());                     /* Display */
    }
  }
  f_in.close();                                       /* Close the file */
```

This example is a code that opens the **gzip-compressed text file foo.txt.gz stored at the Web server** http://www.foo.bar/ and while extracting the file displays only the lines that begin with an alphabetic character. SLLIB also enables editing of character strings and **matching of regular expressions** —operations that you are familiar with in scripting languages— **to be performed easily**.

Here is the other code that handles multidimensional arrays. This is a code that selects a section, divides by 2 all elements of double-precision floating point type, logs those arrays, and displays them:

```
#include <sli/mdarray.h>
#include <sli/mdarray_math.h>
using namespace sli;

int main()
{
    const double arr0_src[] = {0.02, 0.2, 2.0,  20.0, 200.0, 2000.0,  1, 2, 3};
    mdarray_double arr0, arr1;
    /* Set length of array (x,y) */
    arr0.resize_2d(3,3);
    arr0.put_elements(arr0_src, 3*3);
    /* Select first 2 rows in arr0, and copy it into arr1 */
    arr1 = arr0.sectionf("*, 0:1");
    /* Operate log10() for all elements that are divided by 2 */
    arr1 = log10(arr1 / 2);
    for ( size_t j=0 ; j < arr1.length(1) ; j++ ) {
        for ( size_t i=0 ; i < arr1.length(0) ; i++ ) printf("[%g]", arr1(i,j));
        printf("\n");
    }
```

When this code is executed, you have the following result:

```
[-2][-1][0]
[1][2][3]
```

SLLIB enables a selection of elements using IDL/Python-like expression such as `"*,0:1"` and an operation of all the elements of an array to have operators and mathematical functions applied to it so that codes for operating arrays can significantly be simplified. In addition, 2-d or 3-d pointer array can be automatically generated in array objects, therefore, users can write their code for high performance tools without any unnecessary effort.

As described above, SLLIB is a powerful library that compensates the weaknesses of the C language in applications that frequently occur on a routine basis, and provides the primary features that include:

- APIs that enable users to handle various streams (compressed file, network, etc.) readily and in a unified manner.
- Enhancing the processing of character strings. APIs for regular expressions, string arrays and string associative arrays.
- APIs that enable users to easily handle multidimensional arrays.

  Supports selections of elements using IDL/Python-like expression, operations of operators and mathematical functions for all the elements of an array, functions for basic statistics, automatic generation of 2-d or 3-d pointer arrays, etc.

SLLIB enhances the underlying portion of the C-language development environment so that efforts of coding and debugging are reduced and development efficiencies are improved.

Table 1 shows overview of SLLIB's APIs for streams. You will understand that users can handle various streams in a unified manner.

Class name		stdstreamio	gzstreamio	bzstreamio	httpstreamio	ftpstreamio	pipestreamio	digeststreamio	termlineio	termscreenio	inetstreamio
Member functions for the base class cstreamio											
`open()`, etc.	§8.1.1	§8.2.2	§8.3.1	§8.4.1	§8.5.1	§8.6.1	§8.7.1	§8.8.1	§8.9.1	§8.10.1	§8.11.1
`close()`	§8.1.2	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`read()`	§8.1.3	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`write()`	§8.1.3	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`bread()`	§8.1.4	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`bwrite()`	§8.1.5	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`rskip()`	§8.1.6	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`wskip()`	§8.1.7	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`getchr()`	§8.1.8	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`getstr()`	§8.1.9	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`getline()`	§8.1.10	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`scanf()`	§8.1.11	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`putchr()`	§8.1.12	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`putstr()`	§8.1.13	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`printf()`	§8.1.14	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`flush()`	§8.1.15	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`eof()`, etc.	§8.1.16	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`seek()`, etc.	§8.1.17	§8.2.5	-	-	-	-	-	⇐	-	-	-
The new member functions for the inherited classes											
`eprintf()`		§8.2.3	-	-	-	-	-	-	-	-	-
`eflush()`		§8.2.4	-	-	-	-	-	-	-	-	-
`sync()`		-	§8.3.2	-	-	-	-	-	-	-	-
`content_length()`		§8.2.7	-	-	§8.5.2	§8.6.2	-	§8.8.4	-	-	-
`user_agent.assign()`		-	-	-	§8.5.3	-	-	§8.8.5	-	-	-
`username.assign()`		-	-	-	-	§8.6.3	-	§8.8.6	-	-	-
`password.assign()`		-	-	-	-	§8.6.4	-	§8.8.7	-	-	-
`openp()`, etc.		-	-	-	-	-	-	§8.8.2	-	-	-
`is_write_mode()`		-	-	-	-	-	-	§8.8.3	-	-	-
`set_prompt()`		-	-	-	-	-	-	-	§8.9.2	-	-
`automate_history()`		-	-	-	-	-	-	-	§8.9.3	-	-
`add_history()`		-	-	-	-	-	-	-	§8.9.4	-	-
`clear_history()`		-	-	-	-	-	-	-	§8.9.5	-	-
`stifle_history()`		-	-	-	-	-	-	-	§8.9.6	-	-
`unstifle_history()`		-	-	-	-	-	-	-	§8.9.7	-	-
`read_history()`		-	-	-	-	-	-	-	§8.9.8	-	-
`write_history()`		-	-	-	-	-	-	-	§8.9.9	-	-
`path()`		-	-	-	-	-	-	-	-	-	§8.11.2
`host()`		-	-	-	-	-	-	-	-	-	§8.11.3

Table 1: List of the member functions that the base class cstreamio and its inherited classes provide. The symbol, "⇐", indicates that a member function for the base class is inherited. The member functions with a § symbol indicate that they are redefined or are additionally defined.

1.2 The Reason why SLLIB was created

We said in §1.1 that SLLIB compensates the weaknesses of the C language, and compensating the weaknesses of the C language is one of the purposes for which the C++ standard library was created.

Now, let me ask you a question, have ever seen the code "`cout << "foo" << endl;`"? This description of the code shows how "`printf("foo");`" is written according to the manner of the C++ standard library. There is a reason why this notation was made available to use in the C++ standard library, and it is also a fact that this compensates the weaknesses of the C language[1)]. However, many users seem to feel that it makes no sense redefining the bit shift operators with a totally different meaning, or that it is difficult to presume what action is to occur with the code.

The C++ standard library provides a large number of new APIs (chiefly for algorithms) not found in the libc (the standard library for the C language), while it has incorporated as the specifications for APIs a "new manner" that every user will not find easily acceptable. We suppose that learning this "manner" has proved an obstacle that has caused unexpectedly numerous developers to distance themselves from using the C++.

However, if such an obstacle makes developers unable to leverage the advantages provided in the C++ that help to minimize the efforts of coding, it would be a waste of resources. In fact, the APIs such as the scripting language that we discussed in §1.1 can take on a natural form only when the C++ is used.

For this reason, in order to do something about this "waste of resources" situation, we have decided to create a basic library that is useful and leverages the advantages of the C++ while at the same time following the manner of the libc, rather than to build the APIs with the "new manner." Thus, as we reorganized most of the functions in the libc slowly and steadily, we came up with the basic form of SLLIB which has developed into the library that has various scripting language-like APIs.

1.3 Development policies for SLLIB — Following the manner of the libc and leveraging the advantages of the libc

	C++ standard library	**SLLIB**
Feature	Covers a wide range	Limited to applications that occur routinely
libc-like functions	Excluded	Acquired to a full extent
printf() notations	Excluded	Utilized to a full extent
Types of variable	Numerous unique types	Uses only the types defined in libc
Operators	Strong uniqueness	Minimum uniqueness
Ease of use	Not so good	Good

Table 2: Conceptual differences between the C++ standard library and SLLIB

In §1.2, we explained that the C++ standard library made a break with the significant portion of the manner of the libc. In contrast, SLLIB is the library that takes the course of following the footsteps of the manner of the libc.

Table 2 shows the comparison of SLLIB's concepts with those of the C++ standard library, put together in an easy-to-understand manner. As shown in the table, SLLIB limits its features to the applications that occur frequently and makes the APIs close to the C language-like manner to

[1)] However, Google's coding conventions suggest that both the printf() function and the stream in C++ have advantages and disadvantages. In the end, the conventions recommend that the streams in the C++ standard library are not used.

ensure that what users must learn freshly is kept to a minimum. Therefore, SLLIB **can be used easily even by users who are completely unfamiliar with C++**. Also, the uniqueness of the operators is kept to a minimum so that it is unlikely to occur like in the C++ standard library that users who are not familiar with the library cannot read the codes.

A typical example of the C language-like manner is, more than any others, the format of the printf() function and its variable-length arguments. In the arguments of the printf() function, you can describe quite briefly and easily how strings are converted, concatenated, etc. using a wealth of format specifiers. Recognizing that printf() is the most powerful as expected, SLLIB makes the notations of printf() available for use in various APIs.

For example, as there was a function called `f_in.open()` in §1.1, there is also a function called `openf()`, and, like the arguments of the printf() function in the libc, you can also write as follows:

```
f_in.openf("r", "http://%s/%s", "www.foo.bar", "data/foo.txt.gz");
```

In the string processing of SLLIB, a wealth of the APIs are available that follow the manner and function names seen in string.h, ctype.h, etc. in the libc (e.g. atof() and strchr()), making SLLIB a library accessible to C-language programmers.

Also, SLLIB supplements the libc features that the C++ standard library does not have, so it can help users who already use C++ to further refine their codes. Since SLLIB implements most of the libc functions in the member functions of classes, users can **move** from the conventional libc + C++ standard library, or the mixed codes of "procedures" and "object-orientation," **to** SLLIB + C++ standard library, or the more purely "object-oriented" codes.

1.4 All you need is the knowledge of the C language

Reading the sections through to the above, you may have realized that SLLIB has been written using C++. Now, if you think "No way! I never use C++!!", you don't need to worry. **C++ is upward-compatible with C, so when you write "`#include <stdio.h>`", "`printf(...);`" etc. like the C-language codes as you previously did, you can compile the codes with the C++ compiler**. Using C++ does not mean that you need to follow the manner of the C++ standard library, "`cout << "foo" << endl;`".

As we discussed in §1.3, SLLIB is designed to be able to utilize the manner of the C language to a maximum extent. Obviously, the C++-like aspects of SLLIB are addressed by making the descriptions in the manual similar to the C language, so that even users who have no experience in using C++ can easily understand how to use it. Therefore, SLLIB can be easily used by anyone who has knowledge of the C language. Please use SLLIB and find how easy it is like scripting languages to use it.

1.5 What is object-orientation for end-users?

As we discussed a bit about it in §1.3, have you ever heard of the term "object-orientation"? The "object-orientation" approach is useful when realizing in the C language the implementation of scripting language variables, etc. Many books of the world describe the "object-orientation for library developers" using the terms such as "modeling" and "message passing," but it is sufficient for end-users who simply use the library to understand that "this approach helps minimize the efforts of coding."

In this chapter, we will discuss object-orientation focusing on the knowledge that end-users are required to have. The descriptions proceed based on what you have experienced, so don't be afraid of reading the chapter.

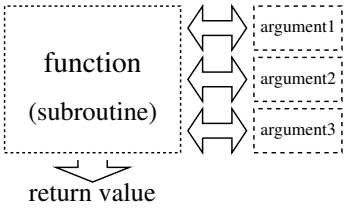
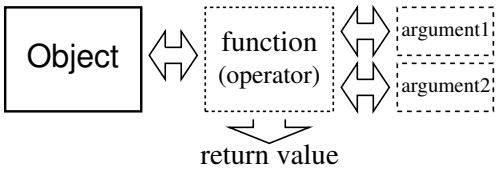
	Procedural	**Object-oriented**
Conceptual diagram		
Example for the C	`fprintf(fp,"foo");`	`a += b;`
Example for FORTRAN	`call sub(x,y,z)`	`x .gt. y`

Table 3: Conceptual diagram of the procedural and object-oriented types, as well as examples of coding manner for each of the types in the C language and FORTRAN. Most languages use both the "procedural" and "object-oriented" types of writing manner as a grammar.

1.5.1 Object-orientation is nothing special

SLLIB is an object-oriented library. Looking at the term of object-orientation, you might think that we are going to discuss a challenging topic. But there is no need to worry at all. The reason is that the object-oriented writing manner appears also in C and FORTRAN. Undoubtedly you already have experience of writing object-oriented codes.

Table 3 shows the conceptual diagram of the "procedural" and "object-oriented" types, and examples of codes in the C language and FORTRAN. As shown in the examples, both the C language and FORTRAN have both the "procedural" and "object-oriented" types of grammar. The "procedural" type is a "command-driven" writing manner as seen in commands such as calling of functions (subroutines), while the "object-oriented" type, although it has the common operation of calling functions, gives the major object of processing at a given time a special grammatical treatment. When you move one of the arguments for the "procedural" type in the conceptual diagram to the left of the function, you have the same diagram as the object-oriented type.

Let us take a look at the examples in Table 3. In the examples for the "procedural" type, that is, "`fprintf(fp,"foo");`" and "`call sub(x,y,z)`", either of the arguments for the function (subroutine) might have the major object of the processing, but all the arguments are grammatically equal. On the other hand, in the examples for the "object-orientated" type, that is, "`a += b;`"and "`x .gt. y`", the major object of the processing is always the variable located to the leftmost, and also is grammatically given a special treatment.

Thus the languages with a high frequency of using the "object-orientation type" of writing manner that clearly defines a major object of processing (object) are called an "object-oriented language," and, on the contrary, the languages with a high frequency of using the "command-driven" writing manner as seen in "`fprintf(fp,"foo");`" are called a "procedural language." The C language and FORTRAN are classified as a "procedural language" because they eventually create and combine functions (subroutines) to achieve the processing.

Then, what kind of language is an "object-oriented language"? Should you be writing only the symbols and abbreviations such as "`+=`" and "`.gt.`" in all occasions? That is not the case. The "object-oriented languages" such as C++ have the extended language specifications to enable the codes to be written in the format of ".string (argument ...)" by generalizing the parts like these "`+=`" and "`.gt.`" so you can write the following code [2]:

```
a.add(b);           /* Add b to a */
```

This is the typical manner of writing codes in object-oriented languages. You might notice that

[2] In reality, the code may not necessarily be able to be written.

it is quite similar to the case of writing "`a += b;`". The major object of the processing, "`a`", is placed at the beginning of the string, and is followed by the verb. Thus object-oriented languages refer to a language in which codes are written mainly using the format of ".string (argument ...)", which is a generalization of the writing manner of "`+=`" and "`.gt.`" A look at this basic concept reveals that object-oriented languages are not a special language. The only difference is that you mainly use the concepts on the right side of Table 3.

In the scripting language of Perl, there are also the symbols such as "`=~`" and "`.=`". You may have used them before. Obviously, these are in an object-oriented writing manner, and have the major object of the processing located to the left. Depending on the language, symbols such as "`#=`" and "`;=`" might possibly exist. However, users who do not have knowledge of the language specifications may not necessarily understand the meaning of these symbols. This lets us to understand that the writing manner of object-oriented languages as in "`a.add(b);`" also provides a superior approach in securing the readability of codes.

1.5.2 Benefits of object-orientation

Thus writing codes in an object-oriented manner makes the major object of processing clear and the codes readable. Moreover, writing codes in an object-oriented manner provides greater benefits.

In C, for the "types" only the codes of fixed byte length such as int and double were able to be used. For that reason, when you write variable length strings or variable length arrays, you must use `malloc()`, `realloc()` and `free()`to operate, secure and release the area, and as a result you have too large codes for the processing you want. As the areas that are secured are accessed using the pointer variable, any of you must have experienced the situation in which accessing the outside of the area resulted in a segmentation violation. The most annoying thing would be a memory leak caused by a missing `free()` statement. If the "types" have a mechanism equipped with it that manages variable length strings and arrays, basically all of these problems are solved. Certainly, object-orientation is here to achieve this. It is for this reason that C++ has the extended language specifications to enable the format of ".string (argument ...)" to be used by generalizing the writing manner of "`=`" and "`+=`", and by library developers designing the "`add()`" part of "`a.add(c);`" that appeared in §1.5.1, users will be able to handle variable length strings and arrays more easily and safely. For example, when you concatenate the two strings, `"abc"` and `"012"`, using SLLIB, you can write as follows:

```
tstring str;
str.printf("abc");          /* Assign "abc" to str */
str.append("012");          /* Add "012" to str */
```

Fortunately, all of the processing of securing the area using `malloc()` and `realloc()` is performed by `.printf()` and `.append()`. Furthermore, the area that are secured by `malloc()`, etc. is automatically released when `str` disappears, so you do not need to worry about memory leak. This means that users simply write what they want to do following the object str, allowing them to focus their thoughts on the content of the processing. As a matter of course, this makes the volume of the codes overwhelmingly small.

These are not all the benefits that object-orientation provides. The greatest benefit is that **hat users should learn can be minimized**. Object-oriented languages use the rule called "inheritance" to disable library developers to create APIs of non-unified specifications. In APIs for SLLIB's stream input/output, "inheritance" plays a highly active role, and when you are able to use an API of a specific type of stream, then you can also use the other types of streams in exactly the same manner.

Here is an example of user codes that uses SLLIB. On the left is a code that opens and displays `foo.txt`, and on the right is a code that opens and displays `foo.txt.gz` while extracting it.

```
#include <sli/stdstreamio.h>
using namespace sli;

int main()
{
    stdstreamio f_in;
    int status = -1;
    char buf[256];
    /* Open the file */
    status = f_in.open("r", "foo.txt");
    if ( status != 0 ) goto quit;
    /* Read and display lines one by one */
    while ( f_in.getstr(buf,256) != NULL ) {
        printf("%s",buf);
    }
    f_in.close();
 quit:
    return status;
}
```

```
#include <sli/gzstreamio.h>
using namespace sli;

int main()
{
    gzstreamio f_in;
    int status = -1;
    char buf[256];
    /* Open the file */
    status = f_in.open("r", "foo.txt.gz");
    if ( status != 0 ) goto quit;
    /* Read and display lines one by one */
    while ( f_in.getstr(buf,256) != NULL ) {
        printf("%s",buf);
    }
    f_in.close();
 quit:
    return status;
}
```

The parts that differ between the left and right are underscored. There are only the three locations where there is a difference between the left and right. The codes do not have any difference at all in how the APIs such as `getstr()` are used. Simply put, that users are able to use standard input/output automatically means in SLLIB that users also are able to use compressed files and files on a network.

Thus object-orientation provides the advantage of enabling users to learn a number of APIs one after another once they learn one API.

1.5.3 Definitions of the terms and conception on codes

In object-oriented languages such as C++, what is formerly referred to as "type" is called "class," and what is formerly referred to as "variable" is called "object" or "instance." And the "`append()`" part of `str.append("012");` as appeared in §1.5.2 called a "member function."

SLLIB is an object-oriented library. Therefore, when you, the user, use the APIs of SLLIB, you first create the objects (instances) that are the major object of processing and use the member functions (`printf() and append()` as in the examples above) in the classes to create the codes. When writing codes for the conventional procedures, you needed to suppose "do something, to something" because the verb (command) must precede the object. In object-oriented languages, you should just instinctively suppose "to something, do something" and simply write it down into the codes.

2 Installation

2.1 Supported operating systems

The operating systems that SLLIB supports include Linux, FreeBSD, MacOSX, Solaris and Cygwin. All of these operating systems support both the 32 bit versions and 64 bit versions.

The compiler required is GCC g++ version 3 series or higher, or Intel ® C++ Compiler (The author of this document used g++ 3.3.2 or higher to verify the capabilities of the system operation).

2.2 Building and installing SLLIB

SLLIB provides two methods for building. Choose one of the following:

- **Method 1**—A method using just only make (§2.2.1)

  Ordinary programmers and science researchers recommend this method.

  This method allows the change of compilers and paths of a library, but basically it assumes that users use the default preferences to build libraries, and is not designed to set detailed preferences.[3)]

  Installs only static libraries. When you need a shared library to be installed, use Method 2.

- **Method 2**— A method using configure and make (§2.2.2)

  This method is recommended when you are a professional software developer or hacker, or when you need to set detailed preferences.

  You can use the options of configure to indicate that zlib, bzip2 or readline is not used.

  Installs both static and shared libraries.

  When you use an operating system other than those supported, try to use this method.

In either of the methods above, the build system automatically detects the operating system you use, and gives the compiler appropriate options to build.

2.2.1 Method 1—A method using just only make

The zlib, bzlib, ncurses and readline libraries are required for building, and must be installed in advance. (In many cases, the name for rpm can be zlib-devel, bzip2-devel, ncurses-devel, readline-devel, etc.[4)] The following is an example for the Redhat series:

```
# yum install zlib-devel
# yum install bzip2-devel
# yum install ncurses-devel
# yum install readline-devel
```

Expand the archive and make it.

```
$ gzip -dc sllib-x.xx.tar.gz | tar xvf -
$ cd sllib-x.xx
$ make
```

Here, when you want to use Intel ® C++ Compiler or to create a library for 64-bit (or 32-bit), you can write as follows, to change the compiler or to add the options for the compiler:

```
$ make CXX=icc
```

[3)] Editing config.h allows you to enable/disable external libraries.

[4)] For the Debian series, the package name is zlib1g-dev, libbz2-dev, ncurses5-dev or libreadline5-dev.

```
$ make CCFLAGS="-m64"
```

If you use 32-bit OS, gcc might not turn SSE2 on by default[5]. You can append options for SSE2 to improve performance as follows:

```
$ make CCFLAGS="-msse2 -mfpmath=sse"
```

Also, when you need to change PREFIX, you can give the value in like manner. For example, do as follows:

```
$ make CCFLAGS="-msse2 -mfpmath=sse" PREFIX="/home/guest/local"
```

Install it.

```
$ su
# make install32
```

This example is for the 32-bit operating system. For the 64-bit operating system, it should be "make install64". By default, for "install32" `libsllib.a` is installed at `/usr/local/lib`, and for "install64" it is installed at `/usr/local/lib64`. Concurrently, the set of header files is copied to `/usr/local/include/sli`, and the wrapper script `s++` of the C++ compiler is installed at `/usr/local/bin`.

2.2.2 Method 2—A method using configure and make

The zlib, bzlib, ncurses and readline libraries are required for building the SLLIB that has all the classes available to use, and must be installed in advance. (In many cases, the name for rpm can be zlib-devel, bzip2-devel, ncurses-devel, readline-devel, etc.[6] The following is an example for the Redhad series:

```
# yum install zlib-devel
# yum install bzip2-devel
# yum install ncurses-devel
# yum install readline-devel
```

Expand the archive, and configure and make it.

```
$ gzip -dc sllib-x.xx.tar.gz | tar xvf -
$ cd sllib-x.xx
$ sh configure
$ make
```

As the options for `configure`, "`--disable-readline`" "`--disable-bz2lib`" "`--disable-zlib`" are available. However, when these options are added, you cannot use the classes that are dependent on the library.

Install the library.

```
$ su
# make install
```

By default, the library files are copied to `/usr/local/lib`, and the set of header files are copied to `/usr/local/include/sli`, and the wrapper script `s++` of the C++ compiler is installed at `/usr/local/bin`.

For the 64-bit operating system, you may need to specify the path for the library using configure, as follows:

[5] It is enabled by default on 64-bit OS.

[6] For the Debian series, the package name is zlib1g-dev, libbz2-dev, ncurses5-dev or libreadline5-dev.

```
$ sh configure --libdir='${prefix}/lib64'
```

3 Tutorial

3.1 Hello World

Here is a familiar program, Hello World.

```
#include <sli/stdstreamio.h>
using namespace sli;

int main()
{
    stdstreamio sio;                /* Create the object (for standard output) */
    sio.printf("Hello World\n");    /* Write to standard output */
    return 0;
}
```

Let us create the above code using the `s++` command.

```
$ s++ hello.cc
```

Following this, you are asked whether you want to create the template code, and then answer "y" to create the code.

Compile and execute the code. When you use `s++`, you can compile the code easily.

```
$ s++ hello.cc
g++ -I/usr/local/include -L/usr/local/lib -Wall -O -o hello hello.cc -lsllib -lz
 -lbz2 -lreadline -lcurses
$ ./hello
Hello World
```

Thus, `s++` enables the `-o` option for the C++ compiler to be automatically added when users do not specify the output file.

In addition, when `s++` has the "/" option specified to it, the program is executed immediately after compiling the code.

```
$ s++ hello.cc /
Hello World
```

When you add arguments following "/", they are given to the program as an argument.

Thus, `s++` enables you to create user programs as though you were using a scripting language.

3.2 Opening and reading files

3.2.1 When standard streams are used

Like in Hello World, use the stdstreamio class (§8.2). This time, use the open() member function (§8.1.1) to open the file for reading.

```
#include <sli/stdstreamio.h>
using namespace sli;

int main()
{
    stdstreamio sio, fin;                 /* Create the object */
    char buf[512];
    if ( fin.open("r","foo.txt") < 0 ) {  /* Open foo.txt as read-only */
        Error handling
    }
    fin.getstr(buf,512);                  /* Read the first line and put it in buf */
    sio.printf("%s",buf);                 /* Output it to the standard output */
    fin.close();
    return 0;
}
```

In SLLIB, `open()` is used instead of `fopen()`, and `getstr()`is used instead of `fgets()`.

Additionally, in SLLIB there is a member function called `getline()` (§8.1.10). This is a function that reads lines up to a newline character `'\n'` and stores them into a buffer inside the object to acquire its beginning address, enabling users to handle text files almost as though they were using Perl. The following example is a code that displays all the content of a text file using `getline()`(The error handling with `open()` is omitted).

```
    const char *line;
    fin.open("r","foo.txt");
    while ( (line=fin.getline()) != NULL ) {
        sio.printf("%s",line);
    }
```

Please note that the buffer area to be returned using `getline()`is managed inside the object, and must not released by users at their discretion (Even if you try to release the area using `free()`, the code cannot be compiled).

Combining the `getline()` and tstring classes (§9.5) enables users to easily edit strings on a per-line basis and match patterns using regular expressions, so that users can analyze text files quite readily. The descriptions for this feature are provided in §3.3.3, so please also see the section.

3.2.2 When the most powerful "versatile" streams are used (Strongly recommended)

Next, we will use the digeststreamio class (§8.8) that supports compressing and extracting of files, networks and pipes. **The digeststreamio class is the most powerful class for stream input/output, and should be strongly recommended**.

```
#include <sli/stdstreamio.h>
#include <sli/digeststreamio.h>
using namespace sli;

int main()
{
    stdstreamio sio;                        /* Create the object (Standard output) */
    digeststreamio fin;                     /* Create the object (File input) */
    char buf[512];
    if ( fin.open("r","foo.txt.gz") < 0 ) { /* Open foo.txt.gz as read only */
        Error handling
    }
    fin.getstr(buf,512);                    /* Read the first line and put it in buf */
    sio.printf("%s",buf);                   /* Output to the standard output */
    fin.close();
    return 0;
}
```

The code opens the gzip-compressed file, `foo.txt.gz`, and reads the extracted content using `getstr()`. The suffix of a path name specified when opening the file is used to automatically determine whether or not it needs to be extracted. Also, when you want to access files through a network, you just simply write:

```
    if ( fin.open("r", "http://www.foo.bar/data/foo.txt.gz") < 0 ) {
```

When a path has `http://` at the beginning of it, the code connects to a Web server, and, when it is determined based on the analysis results of the MIME header and the suffix of the path that the file needs to be extracted, reads the file utilizing zlib or bz2lib.

The following is an example of writing into an FTP server. A user and password can be specified. When a user and password are omitted, the server is accessed anonymously. Compression is also performed depending on the suffix of a path.

```
    fout.open("w", "ftp://username@passwd:ftp.foo.bar/data/foo.txt.gz")
```

Using `openf()` instead of `open()`, you can set the path name in a similar manner to using the arguments for the printf() function. (other classes also have `openf()`)

```
   for ( i=0 ; i < N ; i++ ) {
       fin.openf("r", "ftp://foo.ac.jp/file_%d.txt.gz", i);
         :
       fin.close();
   }
```

Thus, `openf()` is convenient when handling numbered file names or when specifying arguments for a cgi script on a Web server.

Unlike the stdstreamio class, the digeststreamio class has a member function called `openp()` (§8.8.2). The `openp()` member function is a member function specific to the digeststreamio class that can be used in a similar manner to Perl. For example, `fin.open(...)` in the first example can be written as follows:

```
    fin.openp("< foo.txt.gz")
```

`openp()` is extremely useful as it actually allows a wider variety of usages.

The following is an example of inputting using a pipe:

```
    pin.openp("cat /etc/hosts | egrep -v -e '^#' -e '^$' |")
```

The following is an example of outputting using a pile:

```
pout.openpf("| %s", pager)
```

3.2.3 Correspondence relationships with the functions of the libc

`openp()` and `openpf()`that appeared in §3.2.2 are the member functions specific to the digeststreamio class, but `getstr()` and `getline()`, etc. can be used for the digeststreamio class in exactly the same manner as for the stdstreamio class, so that users do not need to learn new APIs. What is more, most of the member functions for the stream input/output in SLLIB all correspond to the standard functions of the libc, so they can be used without feeling challenged. Table 4 shows the correspondence relationships between the stream input/output APIs of the libc and SLLIB.

libc	SLLIB	
`fopen(path, mode)`	`open(mode, path)`, `openf(mode, path_fmt, ...)`	§8.1.1
`fclose(fp)`	`close()`	§8.1.2
`fread(buf, size, nmemb, fp)`	`read(buf, size)`	§8.1.3
`fwrite(buf, size, nmemb, fp)`	`write(buf, size)`	§8.1.3
`fgetc(fp)`	`getchr()`	§8.1.8
`fgets(buf, size, fp)`	`getstr(buf, size)`	§8.1.8
`fscanf(fp, format, ...)`	`scanf(format, ...)`	§8.1.11
`fputc(ch, fp)`	`putchr(ch)`	§8.1.12
`fputs(buf, fp)`	`putstr(buf)`	§8.1.12
`fprintf(fp, format, ...)`	`printf(format, ...)`	§8.1.14
`fflush(fp)`	`flush()`	§8.1.15

Table 4: Correspondence relationships between the stream input/output APIs of the libc and SLLIB

SLLIB has the different specifications for the function names and some of the arguments from those in the libc, and this is because the inconsistency of the APIs seen in the libc has been resolved. For example, looking into the function names of the libc reveals that the libc has the ill-defined naming conventions for function names. In the libc, the functions that contain the meaning of "character" include `getchar()`, `fgetc()` and `strchr()`, and their multiple abbreviations such as "`char`", "`c`" and "`chr`" coexist, making it difficult for the function names to be remembered. Concerning these parts, SLLIB's naming conventions state:

> "character" is abbreviated as "`chr`", "string" is abbreviated as "`str`"

And determine the member function names according to this convention. In addition to the member functions listed in Table 4, the APIs for strings (Table 18)), etc. have this convention applied to them, so SLLIB's member function names as a whole should be able to be easily remembered.

When you are able to use the standard input/output stream (the stdstreamio class; §8.2) or the versatile input/output stream (the digeststreamio class; §8.8) then you can also use the gzip-compressed input/output stream (the gzstreamio class; §8.3), bzip2-compressed input/output stream (the bzstreamio class; §8.4), the http input stream (the httpstreamio class; §8.5), the ftp input/output stream (the ftpstreamio class; §8.6), the pipe input/output stream (the pipestreamio class; §8.7), the per-terminal line input/output (the termlineio class; §8.9), etc. in exactly the same manner. When you want to use these input/output streams, you can specify in the `#include <sli/...>` part a class name you want to use, create the object with a class you want to use, and likewise read and write files, etc. using `open()` and other functions.

The brief summary of the cstreamio class and its derived classes is provided in §7, where Table 6 shows which member function can be used for which class, helping users have an overview of those classes. Please see the section.

3.2.4 Endianness conversion of complex binary data

The bread() member function and bwrite() member function enable binary streams to be read and written while performing the endianness conversion according to the binary data structure defined by the user.

In the following example, a data block of one four-byte integer number, three double types and 64 char types is read twice and stored into the buffer `buf`.

```
stdstreamio f_in;
bstream_info binfo[] = { {4,1}, {-8,3}, {1,64}, {0} };
char buf[512];
ssize_t len;
/* Open the file */
if ( f_in.open("r","binary.dat") < 0 ) {
    Error handling
}
/* Reading of the binary stream (big endian) */
len = f_in.bread(buf, binfo, 2, false);
```

The last argument for `bread()`specifies the endianness of the binary data. In this example, a big endian is specified.

When you define in this code a structure representing the data block, and add the code that assigns the address for the `buf` to the pointer variable, then you can easily access the elements of each data. (In reality, the `binfo` array should be generated from its structure.)

3.2.5 Collaborations with GNUPLOT

When you are doing an operation and you want the results of the operation displayed in a real time, you may usually combine the use of a shell script.

The popen() function of the libc enables commands to be sent from the C language to gnuplot. In SLLIB, you also can do the same thing with the pipestreamio class.

The following code is a code that connects to gnuplot with pipes and sends a draw command to gnuplot to display the undulating animation.

```
#include <sli/pipestreamio.h>
using namespace sli;

int main()
{
    pipestreamio pout;
    int i;

    pout.open("w", "gnuplot");
    pout.printf("set isosamples 48\n");
    for ( i=0 ; i < 1000 ; i++ ) {
        pout.printf("splot sin(%g + sqrt(x*x+y*y))\n", i/10.0);
        pout.flush();
    }
    pout.close();
```

Fig. 1 shows how it appears when the code is executed.

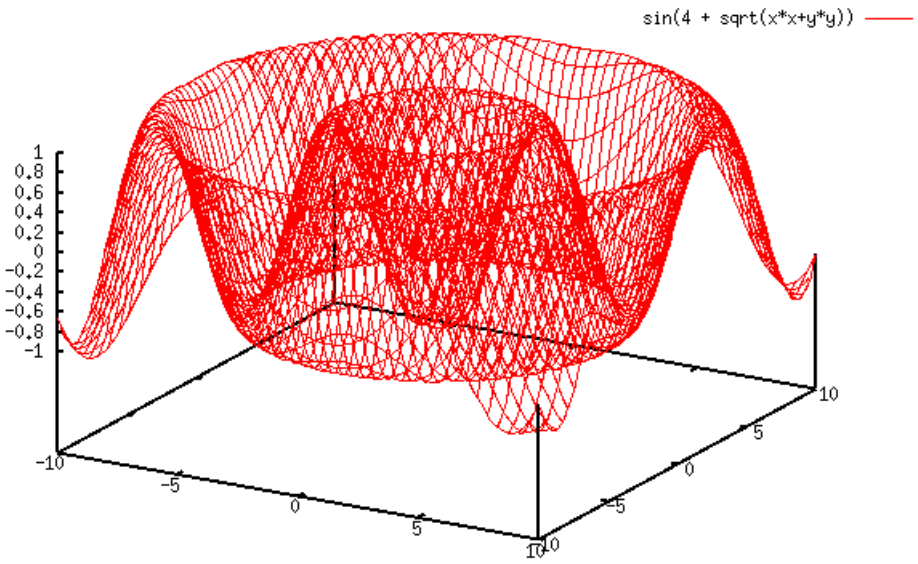


Figure 1: Undulating animation with GNUPLOT

3.3 Operating strings

In this section, we will discuss how strings are operated using SLLIB. Looking over Table 18 to have an overview of what sort of APIs are available will help make it easier to understand the discussions in the section.

3.3.1 Basics

Operating strings in C requires time and efforts. Using the tstring class (§9.5) in SLLIB makes it much easier to operate strings.

First, here is how it seems the class is most typically used.

```
#include <sli/stdstreamio.h>
#include <sli/tstring.h>
using namespace sli;

int main( int argc, char *argv[] )
{
    stdstreamio sio;                                /* Object for the standard output */
    tstring my_str;                                 /* Create the string object */
    my_str = "Hello World";                         /* Assign "Hello World" to my_str */
    sio.printf("my_str = '%s'\n", my_str.cstr());   /* write it into STDOUT */
```

As shown above, strings can be assigned using the "=" operator. "=" for the const char *type enables the address of a string constant to be assigned, while "=" for the tstring class enables the object to have a buffer automatically secured in the inside of it and the entire string to be copied in the buffer. The beginning address of the internal buffer can be acquired using the cstr() member function (§9.5.3). Obviously, the end of the string that is acquired using `cstr()` finishes with `'\0'`, so the string can be swiftly given to `printf()`, etc. as seen above.

And, just all the same, printf() is essential. The printf() member function enables formatted strings to be assigned to an object without having to worry about the size of a buffer.

```
    tstring my_str;                          /* Create the object */
    my_str.printf("filesize is %d", size); /* Assign the result of printf() to my_str */
```

Also in this case, the object has the required buffer automatically secured in the inside of it.

The following is an example of creating a string that combines `argv[1]` and `argv[2]`. This time, the code is written without using the operator "=".

```
int main( int argc, char *argv[] )
{
      :
      :
    tstring my_str;                                  /* Create the object */
    my_str.assign(argv[1]);                          /* Assign argv[1] to my_str */
    my_str.append(argv[2]);                          /* Add argv[2] into my_str */
    sio.printf("my_str = '%s'\n",my_str.cstr());   /* Write it into STDOUT */
    return 0;
}
```

Also in the process of editing strings such as append(), the internal buffer for the object `my_str` is automatically adjusted. (When the object disappears, then the area of the internal buffer also is automatically released.) In this example, the string that combines `argv[1]` and `argv[2]` is put into the internal buffer for `my_str`.

3.3.2 Accessing characters one by one

When you want to read the buffer inside the object one character by one, write as follows:

```
    for ( i=0 ; i < my_str.length() ; i++ ) {
        sio.printf("[%c]",my_str.cchr(i));
    }
```

`length()`returns the length of the string (§9.5.1)), and `cchr(i)` returns the character code (int type) for the `i` th character (§9.5.5).

To edit this buffer with a single character, write:

```
my_str.at(i) = 'a';
```

or

```
my_str[i] = 'a';
```

(§9.5.6, §9.4.1). In this case, the `i` th character of the internal buffer is changed to `'a'`. `i` can be any given value, and if there is not a sufficient area for the internal buffer, the buffer is automatically secured again. The same can be done using the put() member function (§9.5.16).

3.3.3 Applications for reading text files from a stream

The operator "=" (§9.4.2) for the tstring class returns the const char *type to const char * inputs, so that short codes can be written that handle text files from a stream.

The following code is a typical example of how text files are handled in SLLIB.

```
#include <sli/digeststreamio.h>
#include <sli/tstring.h>
    :
    :
    digeststreamio fin;                          /* Object for inputting the stream */
    tstring line;                                /* Use it as a line buffer */
    if ( fin.open("r", "foo.txt.gz") < 0 ) {     /* Open as read only */
        Error handling
    }
    while ( (line=fin.getline()) != NULL ) {
        line.trim("\n");                         /* Delete the newline character */
        /* Various handling */
        sio.printf("%s\n", line.cstr());         /* Display it */
    }
```

In this example, the trim() member function (§9.5.26) is used to delete the newline character, but you can also use the chomp() member function (§9.5.25).

3.3.4 Editing strings

The member functions for editing strings include `append()`, `insert()`, `replace()`, `erase()`, `crop()`, `chomp()`, `trim()`, `toupper()`, `tolower()`, etc. so that addition, insertion, replacement and other operations can be performed easily (§9.5.17 and thereafter).

Many of the member functions have "`f`" as the ending character of their function name. Examples include `insertf()` and `replacef()`, and these can be given an argument to use, like `printf()`.

```
    my_str.insertf(0, "ID:%d ",id); /* Insert the formatted string into the position0 */
```

As appeared in §3.3.3, `trim()`is one of the functions that are used in many occasions.

```
    my_str = "  Hello World  \n";
    my_str.trim(" \n");             /* Delete the white spaces to the right and left */
```

When you write as above, the object `my_str` has `"Hello World"` put into it. (The white space in the center remains.)

In addition to the "`=`" operator that appeared in §3.3.1, the operator "`+=`" is also available to use. "`+=`" is a member function that has the same features as "`append()`", so when you want to combine two strings, you can also write:

```
my_str = argv[1];
my_str += argv[2];
```

3.3.5 Leveraging strings

The member functions for converting strings into other forms include `atof()`, `atoi()`, `scanf()`, etc. (§9.5.35 and thereafter). For example, you write:

```
    double my_value;
    my_value = my_str.atof();
```

and then the string of the internal buffer for `my_str` can be converted into an actual numerical value.

The member functions for comparing or searching for strings include `strcmp()`, `strcasecmp()`, `strchr()`, `strstr()`, `strpbrk()`, `strspn()`, `regmatch()`, etc. (§9.5.41 and thereafter).

For example, to look into whether a string matches the string `"sllib"`, you can write:

```
if ( my_str.strcmp("sllib") == 0 ) {
```

or

```
if ( my_str == "sllib" ) {
```

To compare a string with `"sllib"` case-independently, you can write:

```
if ( my_str.strcasecmp("sllib") == 0 ) {
```

The tstring class has a wealth of member functions. Most of the features provide by the string-related APIs in libc and the member functions of the string class in the C++ standard library are available to use. For details, refer to the reference attached.

3.3.6 Applications of the extended regular expressions (Back reference is also available)

You can search for or replace strings using the POSIX extended regular expressions (the regmatch() member function; §9.5.59, the regreplace() member function; §9.5.30).

First, here is the simplest example. A regular expression is used to look into whether a string is prefixed with a number:

```
stdstreamio sio;
tstring my_str = "123abc";
if ( 0 <= my_str.regmatch("^[0-9]", NULL) ) {
   sio.printf("Matched the string\n");
}
```

When a string is matched, `regmatch()` returns the position of the string that is matched.

Obviously, you can retrieve back reference information. To do so, use the regassign() member function of the tarray_tsring class. See the descriptions in §3.4.8.

This time, a regular expression is used to perform conversion, as though the sed command is used.

```
stdstreamio sio;
tstring my_url = "http://darts.isas.jaxa.jp/foo/";

my_url.regreplace("([a-z]+://)([^/]+)(.*)", "\\2");
sio.printf("hostname = %s\n", my_url.cstr());
```

Using the back reference, the host name "`darts.isas.jaxa.jp`" is retrieved from the URL.

When `regreplace()`has its third argument set with true, it does not just replace a string of the first run but replace the whole string (like `'s/../../g'` in sed).

3.4 Operating string arrays

As shown below, you can create an array of the tstring class in the same manner as normal types.

```
tstring my_arr[32];
my_arr[0] = "abc";
```

However, this way of use does not offer a degree of freedom to edit the arrays.

The tarray_tstring class (§10) provides more flexibility with which to handle string arrays. It enables you to perform various operations that include resizing of arrays, edit strings on any string element (`chomp()`, `trim()`etc.) and search (`find()`, `regmatch()`, etc.), all with much less effort. Obviously, like the tstring class, the size of an internal buffer is automatically adjusted to provide much ease of operation. Acquisition from and conversion to pointer arrays can be done easily to ensure the compatibility with C-language-like codes.

Looking over Table 22 to have an overview of what sort of APIs are available will make it easier to understand the discussions hereinafter.

3.4.1 Immediate assignment

Since the memory management is entirely automatic, you do not need to write codes like "First, 10 objects are secured, and then . . . " Once you have created objects, you can assign them immediately.

The most frequently used methods of substituting objects are substituting a null-terminated pointer array and substituting a constant string, as shown in the following example:

```
#include <sli/stdstreamio.h>
#include <sli/tarray_tstring.h>
using namespace sli;

int main()
{
    stdstreamio sio;
    const char *group1[] = {"sakura", "mizuho", NULL};
    tarray_tstring my_arr = group1;
    size_t i;
    my_arr[2] = "fuji";
    my_arr[3] = "hayabusa";
    for ( i=0 ; i < my_arr.length() ; i++ ) {                    /* Display it */
        sio.printf("%zu: [%s]\n", i, my_arr[i].cstr());
    }
```

The result of the execution is as follows:

```
0: [sakura]
1: [mizuho]
2: [fuji]
3: [hayabusa]
```

You can also assign values when you create objects, as shown below. (Do not forget to add NULL at the end.)

```
    tarray_tstring my_arr("sakura", "mizuho", NULL);
```

For the information about how to initialize objects, refer to §10.1.

3.4.2 Using dprint() for debugging

Inserting `fprintf(stderr, ... );` into codes for debugging is an operation that is commonly seen on a routine basis. However, for arrays you must always take a somewhat cumbersome step of looping the array with a for statement, etc. to display all the elements of the array.

In cases like this, you can output the list of arrays to the standard error output only by using the dprint() member function (§10.4.5). For example, you write:

```
    tarray_tstring my_arr("nEC", "Fujitsu", "Toshiba", NULL);
    my_arr.dprint();
```

and then you have the following displayed:

```
sli::tarray_tstring[obj=0x7fbffff230] = {"nEC", "Fujitsu", "Toshiba"}
```

The address for the object is displayed, which is obviously a value that is dependent on the

environment.

The dprint() member function can also be used in the asarray_tstring class (a class that handles associative arrays; §11) and mdarray_* class (multidimensional arrays; §12). See also §11.4.10 and §12.3.37.

3.4.3 Swiftly passing on to execv() and execvp()

The cstrarray() member function (§10.4.2) enables you to acquire a null-terminated pointer array for the string buffer inside an object, so that it can be simply given to arguments for the char *[] type such as `execvp()` in the libc.

```
tarray_tstring cmd;
  :
execvp(cmd[0].cstr(), (char *const *)cmd.cstrarray());
```

3.4.4 Editing strings on all the elements

For example, you use `split()` as below to split a string in csv format and store it into an array (`split()`is described hereinafter in §3.4.7):

```
const char *line = " Z-80,,  8086 , 6800";
tarray_tstring my_arr;
my_arr.split(line, ",", true);
my_arr.dprint();
```

When this code is executed, the following is displayed:

```
sli::tarray_tstring[obj=0x7fbffff4d0] = {" Z-80", "", "  8086 ", " 6800"}
```

In this case, the white space characters to the right and left of the split strings are unnecessary. When you want to remove these unnecessary white space characters from all the elements, you only need to write as follows:

```
my_arr.trim();
```

The member functions that are capable of operating a string on all the elements in a stroke in this way include `trim()` as well as `chomp()`, `strreplace()`, `regreplace()`, `tolower()`, `toupper()`, etc. (For the complete list, refer to Table 22). For details, refer to the Reference part, §10.4.27 and thereafter.

3.4.5 Editing arrays

As you see `erase()` in the example in §3.4.6, there are also other member functions such as `append()`, `insert()`, `replace()`, etc., which can be used to easily edit arrays in the same manner as in the tstring class (§10.4.16 and thereafter).

The following example shows that elements are inserted and added to the array:

```
tarray_tstring arr;
arr[0] = "SUICA";
arr.insert(0, "ICOCA", 1);                              /* Insert it */
const char *others[] = {"PASPY", "PASMO", NULL};
arr.append(others);                                     /* Append it */
```

The result of this is the arr object that has the four elements contained in it in an order of `"ICOCA"`, `"SUICA"`, `"PASPY"` and `"PASMO"`.

3.4.6 Making arguments for main() easy to use

A well-known argument for `main()` is the getopt() function in the libc, but the tarray_tstring class may be convenient when you do not necessarily need to use `getopt()`.

As shown in §3.4.1, the objects of the tarray_tstring class can have all the string elements of a null-terminated pointer array of the `char *[]` type copied to them using just only the "=" symbol. After the elements are copied, they can be freely edited and applied using the member functions that are provided.

```
#include <sli/stdstreamio.h>
#include <sli/tarray_tstring.h>
using namespace sli;

int main( int argc, char *argv[] )
{
    size_t i;
    double x = 0, y = 0;
    stdstreamio sio;
    /* Make arguments available to use in args */
    tarray_tstring args = argv;

    /* Delete args[0] */
    args.erase(0,1);

    /* Analyze and edit the argument */
    for ( i=0 ; i < args.length() ; i++ ) {
        if ( args[i] == "-x" ) {
            x = args[i+1].atof();
            args.erase(i,2);
        }
        else if ( args[i] == "-y" ) {
            y = args[i+1].atof();
            args.erase(i,2);
        }
    }

    /* Display the arguments that could not be analyzed */
    for ( i=0 ; i < args.length() ; i++ ) {
        sio.printf("%s\n",args[i].cstr());
    }
```

The `==` operator, `.atof()` and `.cstr()` appearing in this example are the member functions of the tstring class. This means that any of the member functions of the tstring class can be used following `args[i]`. Processes using regular expressions can be performed quite easily.

3.4.7 Splitting white space-delimited and CSV-format strings to put into an array—split() member function

The split() member function enables you to split a white space-delimited or CSV-format one-line string into elements and handles them as arrays (§10.4.12). The split() member function provides the remarkably powerful features such as supporting special processing of parts parenthesized with quotation marks, and can switch its behaviors for splitting strings depending on the user's application. The options include:

- Whether or not to allow elements with a string length of zero such as those in CSV format

(When allowed: `"abc,,xyz"` → `"abc", "", "xyz"`) (When not allowed: `"abc  xyz"` → `"abc", "xyz"`)

- To indicate that areas parenthesized with quotation marks and brackets are not allowed to be split (By default, those areas are not given special treatment.)

  (Example: `"abc[ ] 'x y z'"` → `"abc[ ]", "'x y z'"`)

- To specify escape characters (Example: "\"; By default, those characters are not specified.)

  (Example: `"winnt program\\ files"` → `"winnt", "program\\ files"`)

First, the basic form. In this case, the behavior is the same as split in scripting languages.

```
const char *line = "abc def  wxyz ";
tarray_tstring my_arr;
my_arr.split(line, " ", false);             /* Split it into "abc","def","wxyz" */
for ( i=0 ; i < my_arr.length() ; i++ ) {   /* Display the split strings */
    sio.printf("%s\n",my_arr.cstr(i));
}
```

The next shows a case in which strings are in CSV format. The third argument is set to `true` to allow elements with a string length of zero.

```
const char *line = "JAN,,MAR,";
tarray_tstring my_arr;
my_arr.split(line, ",", true);              /* Split it into "JAN","","MAR","" */
```

The following is a case in which parts parenthesized with quotation marks and parentheses are not split.

```
const char *line = "winnt( ) program\\ files 'mary\\'s music'";
tarray_tstring my_arr;
/* Split it into "winnt( )", "program\\ files", "'mary\\'s music'" */
my_arr.split(line, " ", false, "'()", '\\', false);
```

Thus, split() of SLLIB is quite powerful.

3.4.8 Storing the result of regular expression matches

The result of matching regular expressions against a certain string can be stored into an array including back reference information (The regassign() member function; §10.4.13).

The following example shows that a keyword and a value are retrieved for the string `"OS = linux"`.

```
stdstreamio sio;
tarray_tstring my_elms;
tstring my_str = "OS = linux";

my_elms.regassign(my_str, "([^ ]+)([ ]*=[ ]*)([^ ]+)");
if ( my_elms.length() == 4 ) {
    sio.printf("keyword=[%s] value=[%s]\n",
                my_elms.cstr(1), my_elms.cstr(3));
}
```

When this code is executed, it is displayed as "`keyword=[OS] value=[linux]`". The string of a part that matches the whole of `my_pat` is put in `my_elms.cstr(0)` , and the partial string of the back reference is put in `my_elms.cstr(1)` and thereafter.

3.5 Operating associative arrays

The asarray_tstring class (§11)) is a class for handling associative arrays of strings, and can be used quite easily in the same manner as the tarray_tstring class (§11).

Looking over Table 23 to have an overview of what sort of APIs are available will help make it easier to understand the discussions hereinafter.

3.5.1 Immediate assignment

Associative arrays also can be assigned immediately (The operator "[]"; §11.3.1).

```
#include <sli/asarray_tstring.h>
using namespace sli;
      :
    asarray_tstring my_arr;
    my_arr["JR-EAST"] = "SUICA";
    my_arr["JR-CENTRAL"] = "TOICA";
    my_arr["JR-WEST"] = "ICOCA";
    /* Display it */
    for ( i=0 ; i < my_arr.length() ; i++ ) {
        const char *key = my_arr.key(i);
        sio.printf("%s: [%s]\n", key, my_arr[key].cstr());
    }
```

The result of executing the code is as follows:

```
JR-EAST: [SUICA]
JR-CENTRAL: [TOICA]
JR-WEST: [ICOCA]
```

3.5.2 Using dprint() for debugging

Like for the tarray_tstring class, `dprint()` can be used for the asarray_tstring.

You can output the list of associative arrays to the standard error output only by using the dprint() member function (§11.4.10). For example, you write:

```
    asarray_tstring my_arr("PKG","IDL", "VENDOR","ITT", NULL);
    my_arr.dprint();
```

and then you have the following displayed:

```
sli::asarray_tstring[obj=0x7fbffff1f0] = { {"PKG", "IDL"}, {"VENDOR", "ITT"} }
```

The address for the object is displayed, which is obviously a value that is dependent on the environment.

3.5.3 Editing strings on all the elements

Also for associative arrays, the member functions for editing strings on all the elements are available. For example, you can set all the element values lowercase only by writing as follows:

```
    asarray_tstring my_arr("OS","SOLARIS", "VENDOR","Sun", NULL);
    my_arr.tolower();
    my_arr.dprint();
```

Then you have the following displayed:

```
sli::asarray_tstring[obj=0x7fbffff1f0] = { {"OS", "solaris"}, {"VENDOR", "sun"} }
```

The member functions that are capable of operating a string on all the elements in a stroke in this way include `trim()`, `chomp()`, `strreplace()`, `regreplace()`, `toupper()`, etc. (For the complete list, refer to Table 23.) For details, refer to the Reference part, §11.4.26 and thereafter.

3.5.4 Editing

For associative arrays, it is supposedly less likely that the order of elements needs to be addressed. However, since the asarray_tstring class has the implementation requirements that the string arrays are indexed, users can operate the order of the elements. Like for the tarray_tstring class, the member functions such as `append()`, `insert()`, `erase()`are available (§11.4.19 and thereafter).

For example, to insert a set of key and value prior to the key, `"JR-EAST"`, you can write as follows:

```
    my_arr.insert("JR-EAST", "JR-HOKKAIDO","KITACA");
```

3.5.5 Easily accessing data files using split_keys() and split_values()

For example, suppose that you have a data file (with the file name of data.txt.gz) as below:

```
NAME       FROM TO       DISTANCE LOCOMOTIVE     CARRIAGE
Hokuriku   Ueno Kanazawa 517.4    EF64,EF81      14-series
Akebono    Ueno Aomori   772.8    EF64,EF81      24-series
Cassiopeia Ueno Sapporo  1214.9   EF81,ED79,DD51 E26-series
Hokutosei  Ueno Sapporo  1214.9   EF81,ED79,DD51 24-series
```

Let us acquire the content of data.txt.gz into an associative array. When doing this, you can use splits for both the keys and values for the associative array (`split_keys()` ; §11.4.17, `split_values()` ; §11.4.18).

```
#include <sli/stdstreamio.h>
#include <sli/digeststreamio.h>
#include <sli/asarry_tstring.h>
    :
    :
    stdstreamio sio;
    digeststreamio fin;                         /* Object for the stream input */
    tstring line_buf;                           /* Line buffer */
    asarray_tstring vals;
    if ( fin.open("r", "data.txt.gz") < 0 ) {  /* Open as read only */
        Error handling
    }
    if ( (line_buf=fin.getline()) == NULL ) {  /* Read the column name */
        Error handling
    }
    vals.split_keys(line_buf.cstr(), " \n", false);          /* Get the key */
    while ( (line_buf=fin.getline()) != NULL ) {
        vals.split_values(line_buf.cstr(), " \n", false);    /* Get the value */
        sio.printf("%s: %skm\n", vals["NAME"].cstr(), vals["DISTANCE"].cstr());
    }
```

The text data file, data.txt.gz, is opened, and the column name on the first line of the data file is assigned to the key for the associative array using the split_keys() member function. The

data values on the second line and later are set to the values for the associative array using the split_values() member function. Of all the values, the columns for NAME and DISTANCE are output to the standard output.

The result of executing the code is as follows:

```
Hokuriku 517.4km
Akebono 772.8km
Cassiopeia 1214.9km
Hokutosei 1214.9km
```

`split_keys()` and `split_values()`support the processing of double quotation marks and escape characters, like for the tarray_tstring class. For details, refer to the Reference.

3.6 Handling multidimensional arrays without effort

Using the inherited classes for the mdarray class helps you to easily handle the one-dimensional and multidimensional arrays of the types in the C language (§12). The selections of elements using IDL/Python-like expression, mathematical functions (`log10()`, `sin()`, `cos()`, etc. See Table 25), operators (`+`, `-`, `*`, `/`; §**??**) for operating arrays on all their elements, functions for basic statistics, etc. are available. Obviously, users do not need to have codes to secure the memory.

The mdarray class has two operation modes: Users can select the "Auto-resizing mode" that resizes objects automatically, or the "Non-auto resizing mode" that resizes objects based on the user's specifications. These operations modes can be selected when creating objects or when initializing objects using the init() member function. (When nothing is specified, the mode is the "Auto-resizing mode.")

3.6.1 Immediate assignment (Auto-resizing mode: One-dimensional arrays through three-dimensional arrays)

Also for mdarray, you can assign objects immediately.

```
#include <sli/mdarray.h>
using namespace sli;
      :
    mdarray_double arr;                 /* Use it in 8-byte double */
    arr[0] = 1.57079632679489661923;
    arr[1] = 3.14159265358979323846;
    /* Display it */
    for ( i=0 ; i < arr.length() ; i++ ) {
        sio.printf("[%f]\n", arr[i]);
    }
```

This example uses the mdarray_double class, and the other classes include the mdarray_float class, the mdarray_uchar class (the unsinged char type), the mdarray_short class, the mdarray_int class, the mdarray_long class, the mdarray_llong class (the long long type), the mdarray_ssize class (the ssize_t type) and the classes that support several types defined by stdint.h[7)].

For example, to handle arrays of the long type, you write:

```
mdarray_long larr;
```

Also for the Auto-resizing mode, when you want to use an array as a two-dimensional or three-dimensional array, you write:

[7)] The byte size for the short type, the int type, the long type and the long long type is implementation-dependent. When the byte size for a single element needs to be exact, you can use the mdarray_int16 class, the mdarray_int32 class and the mdarray_int64 class.

```
arr(0,1) = 0.31830988618379067154;
arr(0,0,1) = 1.41421356237309504880;
```

and then the array is extended to a two-dimensional or three-dimensional array. For two-dimensional and three-dimensional arrays, "`()`" is used instead of "`[]`" (§12.3.1 and §12.3.2). When the number of dimensions for arrays each is n_0, n_1, n_2, as many internal buffers as $n_0 \times n_1 \times n_2$ are secured. At this time, the value for undefined elements has 0 assigned to it by default, and that default value can also be specified by the user.

If you want to have objects always read using double regardless of the type registered in the object, you can use the dvalue() member function (§12.3.32) and the assign() member function (§12.3.45), as follows:

```
double value;
  :
value = arr.dvalue(x0,y0,z0);
arr.assign(value, x1,y1,z1);
```

Assignments to and operations of objects that are initialized with the integer number type default to "truncating all the decimal places," but can be set to "rounding to the nearest." You can specify which of these two you want to use by using the set_rounding()member function.

3.6.2 Updating number of dimension and elements

Basic method to update number of dimension and elements is to use `resize_1d()`, `resize_2d()`, or `resize_3d()` (§?? and thereafter).

Here is an example:

```
mdarray_double arr;
arr.resize_2d(1024, 768);          /* 2d-array of 1024x768 */
```

3.6.3 Resizing for each dimension

The member functions that are available for the resizing process for each dimension include `resize()`, `resizeby()`, `insert()`, `crop()`, `erase()`, etc. (§12.3.50 and thereafter).

For example, The resize() member function (§12.3.53). resizes objects on a per-dimension basis, as follows:

```
arr.resize(0, 1024);            /* One-dimensional */
arr.resize(1, 768);             /* Two-dimensional */
```

In addition, you write:

```
arr.insert(1, 128, 256);
```

and then the position 128 in the two-dimensional object (The index for dimensions also begins with 0) can have 256 elements inserted into it. The parts in which elements are inserted are initialized.

3.6.4 Operations on arrays

The operators, "`+`", "`-`", "`*`", "`/`", "`+=`", "`-=`", "`*=`", "`/=`", can be used for the objects or scalar values for the mdarray class (§12.3.3 and thereafter). Most of the mathematical functions defined by `math.h` in the libc can be used for the objects or scalar values for the mdarray class (Table 25). The manner of use such as `arr=pow(arr,2.0);` is possible.

The types of the multiple objects used in the operation do not need to be matched. For example, you can operate as follows:

```
mdarray_long larr;
mdarray_double darr;
  :
darr *= 10.0;
darr = log10(darr + larr);
```

In this case, the larr object that handles the long type and the darr object that handles the double type are created, and all the elements for the darr object are multiplied by 10.0, and finally all the elements of darr appended with larr are logged, and the result is assigned to darr.

As shown at the end of this example, the type of the result of operating two objects with different types is the same as when normally operating scalar values.

3.6.5 Non-auto resizing mode (For image buffers)

Like image buffers, applications that the automatic resizing does not suit may exist. In that case, you can specify `false` to the first argument when initializing an object. (Concurrently, you can specify the initial size for each dimension.)

The following example shows that three 1920 x 1080 images where one of the elements is unsigned char (8-bit).

```
mdarray_uchar arr(false);
arr.resize_3d(1920,1080,3);
```

Also in this case, each element is accessed as in:

```
arr(x,y,z) = value;
```

but specifying values outside the range to `x`, `y`, `z` does not result in errors. When a value is out of the range and when you write the value, it is simply discarded, and when you read the value, `INDEF_UCHAR` is returned.

When you want to change the number of dimensions along the way, utilize the increase_dim() member function (§12.3.55) and the decrease_dim() member function (§12.3.56), and when you want to resize the buffer, utilize `resize_1d()`, `resize_2d()`, and `resize_3d()` or the resize() member function (§12.3.53), the resizeby() member function (§12.3.54), etc.

We do not recommend that using both auto resizing mode and non-auto resizing mode in an application, since "`=`" operator copies not only elements but also such attributes.

3.6.6 Fastest access to array elements

The code `arr(x,y,z) = ...;` shown in §3.6.1 and §3.6.5 is safe but not very fast,[8)] since values of arguments are always tested to prevent buffer overrun. If you want fastest access to array elements, the pointer arrays for 2-d or 3-d data are available that are automatically generated in the mdarray object. Use `array_ptr_2d()` or `array_ptr_3d()` (§**??** and §**??**) to obtain a pointer array like this:

[8)] This is not slower than mathematical functions such as `sin()`, etc.

```
mdarray_float arr0(false);
arr0.resize_2d(8,4);
float *const *arr0_ptr = arr0.array_ptr_2d(true);
size_t i, j;
for ( i=0 ; i < arr0.row_length() ; i++ ) {       /* Y */
    for ( j=0 ; j < arr0.col_length() ; j++ ) {   /* X */
        arr0_ptr[i][j] = 100 + 10*i + j;
    }
}
arr0.dprint();
```

The result of executing the code is as follows:

```
sli::mdarray[obj=0xbffff4f0, sz_type=-4, dim=(8,4)] = {
 { 100, 101, 102, 103, 104, 105, 106, 107 },
 { 110, 111, 112, 113, 114, 115, 116, 117 },
 { 120, 121, 122, 123, 124, 125, 126, 127 },
 { 130, 131, 132, 133, 134, 135, 136, 137 }
}
```

The value of argument of `array_ptr_2d()` and `array_ptr_3d()` is a switch to turn on/off for the pointer array generation in the object. Setting `false` will disable it.

The mdarray class manages an internal buffer for an n-dimensional array on a 1-d buffer. Therefore, you can obtain first address of the internal buffer using `array_ptr()` (§12.3.38), and access each element after calculating address of it in your code.

3.6.7 Copy and operation of images using IDL/Python-like expression

Using IDL/Python-like expression, you can easily copy a part of an image to the copy buffer, and paste the image in that copy buffer to any given position. Also, image operations of addition, subtraction, multiplication and division can be performed only by using member functions, and complex image operations can also be performed by collaborating with statistical functions (§**??** and thereafter). Here, we will discuss the simple example for them continued from §3.6.6.

Next code copies a part of float-type array `arr0` shown in §3.6.6 into double-type array `arr1` (including type conversion):

```
mdarray_double arr1(false);
arr1 = arr0.sectionf("4:7, *");
arr1.dprint();
```

The result of executing the code is as follows:

```
sli::mdarray[obj=0xbffff2d0, sz_type=-8, dim=(4,4)] = {
 { 104, 105, 106, 107 },
 { 114, 115, 116, 117 },
 { 124, 125, 126, 127 },
 { 134, 135, 136, 137 }
}
```

For "`=`" operator used in "`arr1 = arr0.sectionf(...);`", *shallow copy* is applied when types of input and output are identical. Note that argument `"4:7,*"` and `"(4:7,*)"` are interpreted as **0-indexed**. If you want to use 1-indexed, expression like `"[...]"` is allowed. That is, `"4:7,*"` is equivalent to `"[5:8,*]"`.

Next code pastes array `arr1` onto $2 <= y$ of array `arr0`:

```
    arr0.pastef(arr1, "*, 2:3");
    arr0.dprint();
```

The result is as follows:

```
sli::mdarray[obj=0xbffff4f0, sz_type=-4, dim=(8,4)] = {
 { 100, 101, 102, 103, 104, 105, 106, 107 },
 { 110, 111, 112, 113, 114, 115, 116, 117 },
 { 104, 105, 106, 107, 124, 125, 126, 127 },
 { 114, 115, 116, 117, 134, 135, 136, 137 }
}
```

When you use `addf()`, `subtractf()`, `multiplyf()` and `dividef()` instead of `pastef()`, you can do image operations of addition, subtraction, multiplication and division. Next code divides elements of `arr0` by that of `arr1` on the same place:

```
    arr0.dividef(arr1, "*, 2:3");
    arr0.dprint();
```

The result is as follows:

```
sli::mdarray[obj=0xbffff4f0, sz_type=-4, dim=(8,4)] = {
 { 100, 101, 102, 103, 104, 105, 106, 107 },
 { 110, 111, 112, 113, 114, 115, 116, 117 },
 { 1, 1, 1, 1, 124, 125, 126, 127 },
 { 1, 1, 1, 1, 134, 135, 136, 137 }
}
```

In member functions that have "`f`" as the ending character of their function name such as `sectionf()`, you can set the argument in a similar manner to using the arguments for the printf() function. For example, you can set a section using some variables like this:

```
    arr1 = arr0.sectionf("%d:%d, %d:%d",
                         x, x + width - 1, y, y + height - 1);
```

3.6.8 Statistics for array elements

SLLIB provides functions to calculate statistics (mean, variance, median, etc.) of elements in mdarray objects. The header file for statistics `sli/mdarray_statistics.h` exists in installed directory (e.g., `/usr/local/include/`), and you can find that the raw code for statistics in the header file, i.e., functions for statistics have 'inline' attributes. This means that you can easily confirm and reuse the routines without using original SLLIB source package.

We show a code to calculate mean, variance, skewness, and kurtosis for last `arr0` in example in §3.6.7:

```
    /* get mean, variance, skewness, kurtosis */
    mdarray_double moment = md_moment(arr0, false, NULL, NULL);
    moment.dprint();
```

The result is as follows:

```
sli::mdarray[obj=0xbfffee90, sz_type=-8, dim=(4)] = {
 87.125, 2657.919355, -0.967679358, -0.8964442873
}
```

Functions only for mean, only for median, etc. are also available, and there are different version of these functions that take statistics of x-direction, y-direction, and z-direction. For example, you

want obtain median[9], the code is written like this:

```
    arr1 = md_median_y(arr0);
    arr1.dprint();
```

The result is as follows:

```
sli::mdarray[obj=0xbffff2d0, sz_type=-8, dim=(8,1)] = {
 { 50.5, 51, 51.5, 52, 119, 120, 121, 122 }
}
```

3.6.9 Combine images

When performing combine of images with mean, median, etc., you can create a 3-d array and use the function to obtain statistics for z-direction. Here is an example code:

```
    /* Prepare a 3-d array for 10 images */
    arr0.resize_3d(1024, 512, 10);

    /* Paste each image onto each plane of 3d array */
    mdarray_float arr_tmp;
    arr_tmp.resize_2d(1024, 512);
    for ( i=0 ; i < 10 ; i++ ) {
           :
        /* some code to prepare contents of arr_tmp here */
           :
        arr0.pastef(arr_tmp, "*,*,%d", i);
    }

    /* Obtain median for z-direction */
    arr1 = md_median_small_z(arr0);
```

When the number of images to combine is large, you can use `md_median_z()` instead of `md_median_small_z()`. `md_median_small_z()` uses temporary buffer for each zx-plane, therefore, it is good for performance but larger memory is required. `md_median_z()` uses temporary buffer for each pixel, therefore, the memory consumption is small. However, it is not good for performance, since there is some overhead of calling internal functions.

3.6.10 Conversion of endianness

The endianness can be converted using the bread() member function and bwrite() member function for the cstreamio class, but when you cannot properly apply these functions, you can use the reverse_endian() member function (§12.3.43) of mdarray to perform the endianness processing.

```
    mdarray_short arr1;
      :
      :
    arr1.reverse_endian(false);
```

To the first argument for `reverse_endian()`, specify `true` when the data to be stored in a file is a little endian, and specify `false` when otherwise. For example, specify `false` for the FITS files that are utilized in astronomy because they are a big endian. Although it is omitted in the example,

[9] SLLIB calculates *true* median using fast algorithm. This is not IRAF's midpt (an approximate value of median).

you can specify to the second and third arguments to perform partial endian conversion of array elements. For details, refer to the Reference.

4 Assumptions that users should comprehend before using SLLIB

Since SLLIB is a library for C++, you the users will use the C++ compilers. C++ basically has upward compatibility with C so you can write codes in the same manner as you did in C, and there is no need to worry.

However, there are several assumptions that users should comprehend before using SLLIB, and they are entirely not difficult to understand. The assumptions include the aspects of C++ that subtly lacks the compatibility with C as a result of extending C++, and the extended features of C++ that are less difficult to use and helpful. These assumptions are discussed in this chapter.

4.1 NAMESPACE

One of the things that are introduced in C++ is a namespace. It is designed to prevent it from occurring that different persons creating functions or types with the same name results in a trouble, and, simply put, it's like a name for category. In SLLIB, a namespace called "sli" is added. For example, when you use some class, formally you use it with `sli::` prefixed, as in "`sli::stdstreamio sio;`". But if you are going to use SLLIB mainly, you will not want to write `sli::` every time. In that case, you write:

```
using namespace sli;
```

and then you can omit `sli::` thereafter. In the examples of use described in this manual, `sli::` is omitted. Users learning C++ for the first time are encouraged to remember that when they write "`#include <sli/...>`", then they should write "`using namespace sli;`".

4.2 NULL and 0

In many processing systems, null for C is defined as:

```
define NULL ((void*)0)
```

However, null for C++ is defined as:

```
define NULL (0)
```

In C++, `NULL` is `0` because in C++ the types of pointer variables are checked more strictly than in C. For example, there are two pointer variables, `char *ptr0;` and `void *ptr1;`, and

```
ptr0 = ptr1;
```

results in an error. However, `0` is defined as "the address of nowhere", so `ptr0 = 0;` does not result in an error. For this reason, `NULL` is `0` in C++.

Now, in C++ the member functions that a class has may have the same name but have different arguments. For example,

```
int foo( int a );
int foo( char *p );
```

are such cases. Here, if you write `hoge.foo(NULL)` or `hoge.foo(0)`, the compiler would not understand which of the functions you want to use. In this case, when you want to use `NULL` or `0`, you must cast it to explicitly indicate the type. Therefore, you must write as follows:

```
hoge.foo((char *)NULL);
hoge.foo((int)0);
```

You can remember securely that in C++, "when you want to use `NULL` **or `0`, you must always cast it**." Or, in another way, you can discard `NULL` and "**cast `0` to use it in all cases**."

4.3 const char *, char *const *, const char *const *

"`const`" is covered by the C language, but unexpectedly many users are not using it properly. Especially, in the pointer variables such as `char *p` and `char **pp`, `const` is important to make the specifications of the functions clear.

In SLLIB, as the arguments and return values for functions, there are expressions such as:

```
const char *p0
char *const *p1
const char *const *p2
```

We will discuss what is `const` in each of these.

For `p0`, changes such as `p0[0] = c;` are forbidden. Operations such as `p0++;` are not forbidden.

For `p1`, changes such as `p1[0][0] = c;` are not forbidden, but changes such as `p1[0] = p;` are forbidden.

For `p2`, changes such as `p2[0][0] = c;` and changes such as `p2[0] = p;` are both forbidden.

s seen above, the specifications of functions are made clear by using `const`.

Also, for other than functions, you will not mistakenly alter areas that are not allowed to be changed by using `const` as follows, for example:

```
const char *name = "My Name";
```

You the users are encouraged to use `const` properly in C or C++ from today on.

4.4 References

References are a new type similar to the pointer type that was introduced in C++, but actually **it can do only the simpler things than the pointer type does**. For that reason, **it can be used easily**.

For example, when you create a reference for the variable called `int a;` with the name of `alias_of_a`, you write:

```
int &alias_of_a = a;
```

References are also called "**alias**", and behave in the manner just as indicated by this name. References are **like symbolic links** in a file system. For example, when you write

```
alias_of_a = 10;
```

, then `a` has `10` assigned to it, and when you write

```
int b = alias_of_a;
```

, then `b` has the value of `a` assigned to it. Also, when you write

```
int *p = &alias_of_a;
```

, then `p` has the address of `a` assigned to it.

As seen above, references are simply designed to provide an "alias" for variables, and do not have a number of "`*`" added to them like pointer variables so that you do not take time identifying them, or do not have `NULL` put in them. References do not have `NULL` put in them because it is forbidden to create a reference of which object does not exist, such as:

```
int &alias_of_a;
```

In SLLIB, references are used when making an object for a structure or class an argument or return value for the member function. The reason why references are used is that since implementing references is to copy an address just the same, making an object for a structure or class an argument using a reference increases efficiency, and "does not let arguments or return values be treated/treat themselves as an array." (For example, when the argument for a function is `foo *p;`, it is not clear whether `p[n]` is accessed or not.) As you may have already understood it, references are called by an address, so when an object is used as an argument for a function, the object referenced may be altered. However, when `const` is attached to the argument, the object will not

be altered. For example, when you have:

```
int strcmp( const tstring &str, size_t pos2 = 0 );
```

, then you can use as follows:

```
tstring foo;
if ( hoge.strcmp(foo) == 0 ) {
```

, and `foo` will not be altered depending on `strcmp()`. (In SLLIB, most of the arguments for references are attached with `const`.)

Unlike pointers, references can do only the simple things, so they can only be used for the limited applications such as handing over arguments for functions as seen above, and are dispensable. Therefore, you the users should not need to use references proactively. You can just remember that when you come across an argument for a reference, you should simply write an object in the argument, without attaching anything to it.

4.5 Pointer variables for an object and arguments/return values for a function

Pointer variables for an object can be created in the same manner as the types in the C language. Here is an example:

```
tstring foo = "abc";
tstring *str_ptr = &foo;
printf("%s\n", str_ptr->cstr());
```

To call a member function from a pointer variable, "`->`" is used instead of "`.`". This rule is the same as the one for structures.

This also applies when using pointer variables as an argument for a function.

```
int my_function( const tstring *str_ptr )
{
    printf("%s\n", str_ptr->cstr());
```

However, when you use a pointer variable as an argument for a function, and when the object for the argument is not altered, make sure that "`const`" is attached as an indication of that.

In C++, you also can use references (§4.4) instead of pointer variables, except in specific cases. References will not have `NULL` put in them, so using references is more secure in this respect. Here is an example:

```
int my_function( const tstring &str_ref )
{
    printf("%s\n", str_ref.cstr());
```

As shown above, for references, "`.cstr()`" is used instead of "`->cstr()`".

You can also make objects a return value for a function.

```
tstring my_function( int a )
{
    tstring ret;
    :
    :
    return ret;
}
```

However, this has some disadvantage in the operation speed, so when you want a speed, you can have an argument for the pointer variable or an argument for the reference returned.

There are some arguments as to whether pointer variables or references should be used as an

argument for a function. For example, Google's conventions state that "when you want to alter an object inside a function (i.e., when it is not const), a pointer variable must be used as an argument." The author of this manual understands that the readers can create rules themselves and follow them.

5 FAQ

5.1 Frequent warnings and errors in compiling

5.1.1 warning: cannot pass objects of non-POD type

When a variable argument is given an object itself, this warning occurs in compiling. Simply executing this mostly results in a segmentation violation.

```
    tstring foo = "abc";
    printf("%s\n",foo);         /* You forgot to attach .cstr() */
```

Add `.cstr()`, etc.

5.1.2 error: 'xxx' was not declared in this scope

Make sure that you have not forgotten to write "`#include <sli/...>`" or"`using namespace sli;`".

5.1.3 error: call of overloaded 'xxx' is ambiguous

This error occurs when the types of the argument on the user program side and on the library side are not completely matched, and it can not be determined which of the member functions the user program is attempting to use.

Especially, you must be careful when you have given `NULL` or `0` to the argument. Remember that in C++ `NULL` is zero itself. For `NULL` and `0`, also refer to §4.2.

When this error occurs, try to cast the argument.

5.1.4 error: invalid conversion from 'const char*' to 'char*'

Writing the code as below results in this error:

```
    tstring foo = "abc";
    char *p = foo.cstr();       /* You forgot to attach const */
```

The second line should be "`const char *p =` ..." Also, you must not cast it forcefully.

5.1.5 error: passing 'xxx' as 'yyy' argument of 'zzz' discards qualifiers

Writing the code as below results in this error:

```
int my_function( const tstring &my_arg )
{
    my_arg.append("abc");
```

The code is attempting to alter the object foo inside the function, but the argument for the function has the "const" attribute, so this results in an error.

6 Information for advanced users

6.1 Instructions for creating objects in the heap

In all the examples of code shown earlier in the manual, objects for a class were created in the stack. That is satisfying in most cases, but there may be cases in which you want to create objects in the heap by any means.

When you want to create objects in the heap, you must be careful. In that case, you cannot create an object using the malloc() function and the realloc() function. You must always use the "new" operator. And objects that are created using the "new" operator must always be deleted using the "delete" operator.

Here is an example of using `new` and `delete` to create and delete an object.

```
tstring *mystr_ptr = new tstring;          /* Create the object */
*mystr_ptr = "abc";                        /* Assign it */
mystr_ptr->append("xyz");                  /* Add it */
printf("%s\n", mystr_ptr->cstr());         /* Display it */
delete mystr_ptr;                          /* Delete the object */
```

You can also give an argument for the constructor, as in "`new tstring(64);`".

Obviously, when you forget to write `delete` after `new`, a memory leak occurs. For this reason, you should minimize the use of `new` and `delete`.

6.2 When you want to create an array of objects in the heap

Also when you want to create an array of objects in the heap, you cannot use the malloc() function and the realloc() function. You must use the tarray template class or the asarray template class described in the SLLIB Advanced Course Manual, or the vector template class, etc. in STL.

The following example is a code that uses the asarray template class that is capable of creating the associative array for any given class. It manages the object for the asarray_tstring class using the associative array:

```
#include <sli/asarray_tstring.h>
#include <sli/asarray.h>
    :
    :
    asarray<asarray_tstring> my_config;
    my_config["TEST"]["HOST"] = "www.foo.com";
    printf("%s\n", my_config["TEST"]["HOST"].cstr());
```

6.3 Collaborations between structures and classes

Classes can also be used to define members for a structure. For example, you can use a class as follows:

```
struct mytag {
  int id;
  tstring name;
};
    :
    :
    struct mytag foo;
    foo.id = 0;
    foo.name = "gates";
    sio.printf("%s\n",foo.name.cstr());
```

Or, you can also use a class as an array, as follows:

```
    struct mytag foo_arr[256];
```

As shown above, when you create an object for a structure from the stack, you do not need to take care at all. However, when you create an object from the heap, you must do so in the manners described in §6.1 and §6.2.

6.4 Handling of the exceptions, try {} & catch ()

SLLIB does not provide exceptions save "failed to secure the memory" and only a few other cases. Therefore, if you do not wish to learn full-fledged coding, you may skip this section.

`try{}` and `catch(){}` are the syntax for handling the "exceptions" that were introduced in C++. SLLIB generates an "exception" when a **fatal problem** occurs which is not related to any of the user-given arguments, such as "**failed to secure the memory**"[10]. This "exception" can be captured using `try{}` and `catch(){}` in the user's code. For SLLIB, exceptions that are generated always have a message of the `err_rec` type, so when you want to capture them, you write as follows:

```
    try {
        /* Resize the buffer for the array */
        array.resize(0,very_big_size);
        return_status = 0;
    }
    catch ( err_rec msg ) {
        sio.eprintf("[EXCEPTION] function=[%s::%s] message=[%s]\n",
                    msg.class_name, msg.func_name, msg.message);
        return_status = -1;
    }
```

Please note that if an exception occurs when you are not using `try{}` and `catch( err_rec msg ){}`, the abort() function is called, and the program terminates.

Normally, when a fatal error such as this occurs, in most cases the program are no longer able to be continued. Therefore, you do not need to use `try{}` and `catch( err_rec msg ){}` if you allow for the specifications that enable the abort() function to be called when an exception occurs.

[10] The new operator described in §6.1 causes the bad_alloc exception to occur when you fail to assign the memory. For the bad_alloc exception, please look into the details on Google, etc.

7 The CSTREAMIO class and a summary of its inherited classes

The cstreamio class is the abstract base class (a class in which the specifications for the basic member functions are formulated) that provides the file input/output APIs that are extremely similar with `stdio.h` in the libc. Understanding the specifications of the member functions for the cstreamio class enables users to handle a variety of streams that are supported by the inherited classes, without learning the APIs again from the scratch.

As shown in Table 5 there are the inherited classes that support the various streams, and the inherited classes have the member functions additionally defined in them that specialize in their respective streams. This section provides a summary of the CSTREAMIO class and its inherited classes, organized mainly on the tables.

For the examples of their use, refer to the examples in §3.2 of the Tutorial or in the Reference (§8.1 and thereafter).

7.1 A summary of the inherited classes

Table 5 shows a list of the cstreamio class and its inherited classes.

	Class name	Features
§8.2	stdstreamio	Standard input/output, standard error output, standard file input/output (corresponds to `stdio.h`)
§8.3	gzstreamio	gzip File input/output that supports compressions and extractions
§8.4	bzstreamio	bzip2 File input/output that supports compressionis and extractions
§8.5	httpstreamio	http Input from servers (download)
§8.6	ftpstreamio	ftp Input from servers (download) and output to ftp servers (upload)
§8.7	pipestreamio	Input from pipes and output to pipes
§8.8	digeststreamio	Automatically switches and uses {std, gz, bz, http, ftp} streamio depending on the path name
§8.9	termlineio	Helpful command input (Wrapper to readline)
§8.10	termscreenio	Input/output to the terminal screens (Starts the pager or editor)
§8.11	inetstreamio	Low-level Internet client for the one- or two-way sequential connection

Table 5: List of the inherited classes of the cstreamio classes that users actually use

The member functions that are common to the inherited classes shown above are described in §8.1 and thereafter, and the member functions that are additionally defined in the inherited classes are described in §8.2 and thereafter. For the overview of those classes, refer to Table 6.

7.2 Overview of the implementation of the member functions for the base classes and inherited classes

Table 6 show a list of the member functions that indicates which of the member functions can be used for each class, or how it is implemented. The member functions on the upper half of the table are those formulated in the abstract base classes cstreamio, and the member functions on the lower half of the table are those additionally defined in the inherited classes.

Class name		stdstreamio	gzstreamio	bzstreamio	httpstreamio	ftpstreamio	pipestreamio	digeststreamio	termlineio	termscreenio	inetstreamio
Member functions for the base class cstreamio											
`open()`, etc.	§8.1.1	§8.2.2	§8.3.1	§8.4.1	§8.5.1	§8.6.1	§8.7.1	§8.8.1	§8.9.1	§8.10.1	§8.11.1
`close()`	§8.1.2	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`read()`	§8.1.3	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`write()`	§8.1.3	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`bread()`	§8.1.4	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`bwrite()`	§8.1.5	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`rskip()`	§8.1.6	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`wskip()`	§8.1.7	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`getchr()`	§8.1.8	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`getstr()`	§8.1.9	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`getline()`	§8.1.10	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`scanf()`	§8.1.11	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`putchr()`	§8.1.12	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`putstr()`	§8.1.13	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`printf()`	§8.1.14	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`flush()`	§8.1.15	⇐	⇐	⇐	-	⇐	⇐	⇐	⇐	⇐	⇐
`eof()`, etc.	§8.1.16	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐	⇐
`seek()`, etc.	§8.1.17	§8.2.5	-	-	-	-	-	⇐	-	-	-
The new member functions for the inherited classes											
`eprintf()`		§8.2.3	-	-	-	-	-	-	-	-	-
`eflush()`		§8.2.4	-	-	-	-	-	-	-	-	-
`sync()`		-	§8.3.2	-	-	-	-	-	-	-	-
`content_length()`		§8.2.7	-	-	§8.5.2	§8.6.2	-	§8.8.4	-	-	-
`user_agent.assign()`		-	-	-	§8.5.3	-	-	§8.8.5	-	-	-
`username.assign()`		-	-	-	-	§8.6.3	-	§8.8.6	-	-	-
`password.assign()`		-	-	-	-	§8.6.4	-	§8.8.7	-	-	-
`openp()`, etc.		-	-	-	-	-	-	§8.8.2	-	-	-
`is_write_mode()`		-	-	-	-	-	-	§8.8.3	-	-	-
`set_prompt()`		-	-	-	-	-	-	-	§8.9.2	-	-
`automate_history()`		-	-	-	-	-	-	-	§8.9.3	-	-
`add_history()`		-	-	-	-	-	-	-	§8.9.4	-	-
`clear_history()`		-	-	-	-	-	-	-	§8.9.5	-	-
`stifle_history()`		-	-	-	-	-	-	-	§8.9.6	-	-
`unstifle_history()`		-	-	-	-	-	-	-	§8.9.7	-	-
`read_history()`		-	-	-	-	-	-	-	§8.9.8	-	-
`write_history()`		-	-	-	-	-	-	-	§8.9.9	-	-
`path()`		-	-	-	-	-	-	-	-	-	§8.11.2
`host()`		-	-	-	-	-	-	-	-	-	§8.11.3

Table 6: List of the member functions that the base class cstreamio and its inherited classes provide. The symbol, "⇐", indicates that a member function for the base class is inherited. The member functions with a § symbol indicate that they are redefined or are additionally defined.

8 References for the CSTREAMIO class and its inherited classes

8.1 Member functions for the CSTREAMIO class

Table 7 6 shows a list of the member functions. For the member functions that have the same feature as in the libc, the functions that correspond are shown.

	cstreamio class	Feature	Corresponding function in libc
§8.1.1	`open()`, `openf()`, `vopenf()`	Opens a stream	`fopen()`
§8.1.2	`close()`	Closes a stream	`fclose()`
§8.1.3	`read()`	Input of a binary stream	`fread()`
§8.1.3	`write()`	Output of a binary stream	`fwrite()`
§8.1.4	`bread()`	Input of a binary stream (With endian conversion)	—
§8.1.5	`bwrite()`	Output of a binary stream (With endian conversion)	—
§8.1.6	`rskip()`	Seek forward (if possible) or read n bytes to skip data stream	—
§8.1.7	`wskip()`	Seek forward (if possible) or write n bytes of blank data	—
§8.1.8	`getchr()`	Input of a character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of a line	—
§8.1.11	`scanf()`	Converts an input with a format and assigns it to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of a character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs a value of an argument after format-converted	`fprintf()`
§8.1.15	`flush()`	Outputs the content of a buffer forcefully	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.1.17	`seek()`	Change position of streams	`fseek()`
§8.1.17	`rewind()`	Changes position of streams to the beginning	`rewind()`
§8.1.18	`tell()`	Value of stream position indicator	`ftell()`
§8.1.19	`is_seekable()`	Test for seekable stream or not	-

Table 7: List of the member functions that the cstreamio class provides

8.1.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Opens a stream

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, cstreamio &sref );  ..... 3
int open( const char *mode, const char *path );  ..... 4
int openf( const char *mode, const char *path_fmt, ... );  ..... 5
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 6
```

DESCRIPTION

Opens the file shown in `path` or `path_fmt`, or connects to a stream the descriptor specified by

`fd` or the object for an inherited class of the cstrreamio class specified by `sref`. As for `mode`, for the member functions 1 and 2, specify `"r"` when reading and `"w"` when writing. Also for the member functions 3 through 6, basically the same applies, but the available values differ depending on the inherited class, so refer to the descriptions for each inherited class.

The member function 3 is used in cases such as the one in which, when an object for an inherited class of the cstreamio class has any stream opened by it, the stream is changed during the course of the stream to a stream that is gzip- or bzip2-compressed (Refer to EXAMPLE-2).

For the member functions 5 and 6, the arguments for `path_fmt` and thereafter can be specified in the same manner as the ones for `printf()` and `vprintf()` in the libc. For the format for `printf()` and `vprintf()`, refer to the descriptions in §8.1.14.

PARAMETER

[I]	`mode`	File opening mode
[I]	`fd`	File descriptor
[I]	`sref`	Object for inherited class of cstreamio class
[I]	`path`	File name
[I]	`path_fmt`	Specifications for file name format
[I]	`...`	Each element of a file name
[I]	`ap`	All the elements of a file name

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination.
Negative value (error)	:	If the system failed to open the stream because the file does not exist etc.
	:	If the system failed to open the stream because the `mode` specified was inappropriate etc.
	:	If the system failed to open the stream because the relationship between the `mode` specified and `fd` is incorrect etc (Member function 2).
	:	If the system failed to open the stream because it cannot access the stream in the specified mode etc.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the member functions detailed in this section.

EXCEPTION

If the system fails to copy a file or descriptor for the standard input/output.

EXAMPLE-1

The following code opens *file.txt* in *directory* in read mode:

```
stdstreamio f_in;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

f_in.close();
```

EXAMPLE-2

The following code reads *complex_file.dat* made up of a one-line text header and gzip-compressed binary data, and then prints it to standard output:

```
stdstreamio f_in_text;
gzstreamio f_in_gz;
char c_buf[256];

if (f_in_text.open("r", "complex_file.dat") < 0) {
    Error handling
}

/* Read and display header */
printf("Header: %s", f_in_text.getline());

/* Read compressed data */
if (f_in_gz.open("r", f_in_text) < 0) {
    Error handling
}

/* Read and display compressed data line by line */
while (f_in_gz.getstr(c_buf, 256) != NULL) {
    printf("%s",c_buf);
}

f_in_gz.close();
f_in_text.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

Refer to the descriptions for the class you actually use with the member functions of 3 to 6. The classes are described below:

§8.2.2	`stdstreamio::open()`	§8.3.1	`gzstreamio::open()`
§8.4.1	`bzstreamio::open()`	§8.5.1	`httpstreamio::open()`
§8.6.1	`ftpstreamio::open()`	§8.7.1	`pipestreamio::open()`
§8.8.1	`digeststreamio::open()`	§8.9.1	`termlineio::open()`
§8.10.1	`termscreenio::open()`	§8.11.1	`inetstreamio::open()`

8.1.2 close()

NAME

close() — Closes a stream

SYNOPSIS

```
int close();
```

DESCRIPTION

Closes a stream opened by `open()`.

RETURN VALUE

0	:	Normal termination
Non-zero	:	Error

EXAMPLE

The following code opens *file.txt* in *directory* in read mode and then closes it:

```
stdstreamio f_in;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

f_in.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.3 read(), write()

NAME

read(), write() — Input/output of streams

SYNOPSIS

```
ssize_t read( void *buf, size_t size );
ssize_t write( const void *buf, size_t size );
```

DESCRIPTION

`read()` reads `size` bytes of data from the stream opened by `open()`, and then stores it in a buffer given by `buf`.

`write()` writes `size` bytes of data acquired from the buffer specified by `buf` to the stream opened by `open()`.

PARAMETER

[I]	`buf`	A buffer used to store data (For `read()`)
[O]	`buf`	A buffer used to store data (For `write()`)
[I]	`size`	Size of data

([I] : Input, [O] : Output)

RETURN VALUE

Positive value	:	Number of bytes successfully read and written.
0	:	If the EOF of a stream to be read is reached. (`read()`)
	:	If the `size` specified is 0. (`read()`)
Negative value (error)	:	If the `buf` specified is inappropriate.
	:	If the stream is not opened. (`read()`)
	:	If open mode for the input/output stream is inconsistent.
	:	If the stream to be read is abnormal before using the member function. (`read()`)
	:	If the system failed to read and write a stream for any other operating environment reason than described above. ⟦For example, if a stream referenced by a file descriptor does not have enough space.⟧

EXAMPLE

The following code reads 512 bytes of data from the file *file.txt* in the directory of *directory*, opens it in read mode, and then stores it in the buffer given by `buf`:

```
stdstreamio f_in;
char c_buf[1024];

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

if (f_in.read(c_buf, 512) < 0) {
    Error handling
}

printf("%s", c_buf);

f_in.close();
```

WARNING

cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

If the size of a buffer specified by `buf` is smaller than `size`, the program may terminate abnormally or be forcefully terminated.

8.1.4 bread()

NAME

bread() — Input of binary streams

SYNOPSIS

```
ssize_t bread( void *buf, ssize_t sz_type, size_t n,
               bool little_endian );  ..................................... 1
ssize_t bread( void *buf, const bstream_info binfo[], size_t n,
               bool little_endian );  ..................................... 2
```

DESCRIPTION

`bread()` reads binary data from the stream opened by `open()`, and then stores it in the buffer given by `buf`. Converts the order of the bytes, depending on the endian specifications provided by `little_endian`, and the operating system for the stream data.

With member function 1, `n` integer numbers or floating-point values of |`sz_type`| bytes are read. With floating-point values, `sz_type` is assigned a negative number.

With member function 2, the structure of binary data can be given by a bstream_info structure array, thus allowing even complex data to be handled. Data blocks defined by array `bstream_info` are read `n` times.

A bstream_info structure is defined as follows:

```
typedef struct {
    ssize_t sz_type;
```

```
    ssize_t length;
} bstream_info;
```

The member `sz_type` is given the number of bytes and type of data (a floating-point value if negative) of an element, while `length` is provided by the number of data. To indicate the end of the definition, `sz_type` must have `0` set at the end of the array.

PARAMETER

[O]	`buf`	Buffer used to store data (For `bread()`)
[I]	`sz_type`	Number of bytes and type of data of element (Negative value: For floating-point values, Positive value: For integer numbers)
[I]	`binfo`	Definition of the structure of a block of binary data
[I]	`n`	Number of data or data blocks
[I]	`little_endian`	Endian specifications for stream data (`true` : Little endian, `false` : Big endian)

([I] : Input, [O] : Output)

RETURN VALUE

Positive value	:	Number of bytes successfully read.
0	:	If the EOF of a stream to be read is reached.
	:	If reading of 0 byte is specified by an argument.
Negative value (error)	:	If the specified `buf` is inappropriate.
	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.
	:	If the stream to be read is abnormal before the member function is used.
	:	If the system has failed to read and write a stream for any other operating environment reason than described above. ⟦For example, if a stream referenced by a file descriptor encounters an abnormal operation.⟧

EXAMPLE

The following code reads 4 bytes of data from the binary file of *file.dat* in the directory of *directory*, and stores it in a buffer given by `ui_buf`. It then prints the content of `ui_buf` to standard output in the hexadecimal format. In this case the binary file *file.dat* described as a little endian.

```
stdstreamio f_in;
unsigned int ui_buf;

if (f_in.openf("r", "%s/%s", "directory", "file.dat") < 0) {
    Error handling
}

if (f_in.bread(&ui_buf, sizeof(ui_buf), 1, true) < 0) {
    Error handling
}

printf("%X\n", ui_buf);

f_in.close();
```

For more examples of using member function 2 refer to EXAMPLE in §8.1.5.

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

If the size of a buffer specified by `buf` is smaller than specified by the argument, the running program may terminate abnormally or be forcefully terminated.

8.1.5 bwrite()

NAME

bwrite() — Output of binary streams

SYNOPSIS

```
ssize_t bwrite( const void *buf, ssize_t sz_type, size_t n,
                bool little_endian );  ..................................... 1
ssize_t bwrite( const void *buf, const bstream_info binfo[], size_t n,
                bool little_endian );  ..................................... 2
```

DESCRIPTION

`bwrite()` writes binary data provided by `buf` to a stream opened by `open()`, and converts the byte order depending on the endian specifications provided by `little_endian` and the operating system for the stream data.

With member function 1, `n` integer numbers or floating-point values of |`sz_type`| bytes are written. With floating-point values, `sz_type` is given a negative number.

With member function 2, the structure of binary data can be provided with a bstream_info structure array, thus allowing even complex data to be handled. Data blocks defined by the `bstream_info` array get written `n` times.

The bstream_info structure is defined as follows:

```
typedef struct {
    ssize_t sz_type;
    ssize_t length;
} bstream_info;
```

The member `sz_type` is given the number of bytes and type of data (a floating-point value if negative) of an element, and `length` is the number of data. To indicate the end of the definition, `sz_type` at the end of the array must have `0` set to it.

PARAMETER

[I]	`buf`	Buffer used to store data
[I]	`sz_type`	Number of bytes and type of data of element (Negative value: For floating-point values, Positive value: For integer numbers)
[I]	`binfo`	Definition of the structure of a block of binary data
[I]	`n`	Number of data or data blocks
[I]	`little_endian`	Endian specifications for stream data (`true` : Little endian, `false` : Big endian)

([I] : Input, [O] : Output)

RETURN VALUE

Positive value	:	Number of bytes successfully written.
0	:	If writing 0 byte is specified by an argument.
Negative value (error)	:	If the `buf` specified is inappropriate.
	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.
	:	If the stream to be written is abnormal before using the member function.
	:	If the system failed to read or write a stream for any other operating environment reason than described above. [[For example, if a stream referenced by a file descriptor does not have enough space.]]

EXCEPTION

If the system failed to secure a temporary buffer for use in converting an endian.

EXAMPLE

The following code writes the data blocks defined by `binfo` once to the binary file of *file.dat* in the directory of *directory*. A data block is defined as having three double types and 32 char types. In this case the binary file *file.dat* is described as big endian.

```
stdstreamio f_out;
bstream_info binfo[] = { {-8,3}, {1,32}, {0} };
char buffer[256];
  :
Register data to buffer
  :
if (f_out.openf("w", "%s/%s", "directory", "file.dat") < 0) {
    Error handling
}

if (f_out.bread(buffer, binfo, 1, false) < 0) {
    Error handling
}

f_out.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

If the size of a buffer specified by `buf` is smaller than that specified by the argument the running program may terminate abnormally or be forcefully terminated.

8.1.6 rskip()

NAME

rskip() — Seek forward (if possible) or read *n* bytes to skip data stream

SYNOPSIS

```
ssize_t rskip( size_t n );
```

DESCRIPTION

To skip data of readable stream, `rskip()` performs a seek forward for seekable stream. If seek is not possible, it reads `n` bytes and throws the data away.

PARAMETER

[I] `n` Byte length to be skipped
([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Byte length successfully sought or read.
negative value	:	Error.

EXCEPTION

If the system failed to secure a temporary buffer for use in reading data stream.

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.7 wskip()

NAME

wskip() — Seek forward (if possible) or write n bytes of blank data

SYNOPSIS

```
ssize_t wskip( size_t n, int ch );
```

DESCRIPTION

To skip data of writable stream, `wskip()` performs a seek forward for seekable stream. If seek is not possible, it writes `n` bytes of blank data.

PARAMETER

[I] `n` Byte length to be skipped
[I] `ch` Blank character to be written
([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Byte length successfully sought or written.
negative value	:	Error.

EXCEPTION

If the system failed to secure a temporary buffer for use in writing data stream.

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.8 getchr()

NAME

getchr() — Input of characters

SYNOPSIS

```
int getchr();
```

DESCRIPTION

`getchr()` reads a character from a stream opened by `open()`, and returns it as the int type.

RETURN VALUE

Non-negative value	:	A value of a read character of the unsigned char type cast to the int type.
EOF	:	If the end of a stream is reached.
EOF (error)	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.

EXAMPLE

The following code opens *file.txt* in *directory* in read mode, reads a character from it, and then prints it to standard output:

```
stdstreamio f_in;
int i_chr;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

if ((i_chr = f_in.getchr()) == EOF) {
    printf("EOF\n");
}
else {
    printf("%c\n", i_chr);
}

f_in.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.9 getstr()

NAME

getstr() — Input of character strings

SYNOPSIS

```
char *getstr( char *s, size_t size );
```

DESCRIPTION

`getstr()` reads a maximum of `size`-1 characters from a stream opened by `open()`, and stores them in a buffer specified by `s`. Reading terminates after reading the EOF or newline character. The newline character read is also stored in the buffer specified by `s`.

PARAMETER

[O]	`s`	A buffer to store the read characters
[I]	`size`	Number of characters to be read

([I] : input, [O] : output)

RETURN VALUE

Non-NULL value	:	Address of storage buffer.
NULL	:	If the end of a stream is reached.
NULL (error)	:	If the specified `buf` or `size` is inappropriate.
	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.

EXCEPTION

If the system fails to secure a buffer for use in reading the data. (`getstr()`)

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.10 getline()

NAME

getline() — Input of characters and character strings

SYNOPSIS

```
const char *getline();  ..... 1
const char *getline( size_t nchars );  ..... 2
```

DESCRIPTION

Reads strings up to the newline character from a stream opened by `open()`in the buffer inside the object, and then returns the address of that internal buffer. This address will be invalid if the member function for the object is called next. If the number of characters needs to be limited specify `nchars`. In this case the characters are read until the newline character appears or the `nchars` nth character is reached.

PARAMETER

[I] `nchars` Limit value for the number of characters to be read
([I] : Input, [O] : Output)

RETURN VALUE

Non-NULL value	:	Address of internal buffer.
NULL	:	If the end of a stream is reached.
NULL (error)	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.

EXCEPTION

If the system fails to secure a buffer for use in reading the data.

EXAMPLE

The following code opens *file.txt* in *directory* in read mode, reads lines one by one from it, and then prints them to standard output:

```
stdstreamio f_in;
const char *line_ptr;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}
```

```
while ((line_ptr = f_in.getline()) != NULL) {
    printf("%s", line_ptr);
}

f_in.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

8.1.11 scanf(), vscanf()

NAME

scanf(), vscanf() — Formatted input conversion

SYNOPSIS

```
int scanf( const char *format, ... );  ..... 1
int vscanf( const char *format, va_list ap );  ..... 2
int scanf( size_t nchars, const char *format, ... );  ..... 3
int vscanf( size_t nchars, const char *format, va_list ap );  ..... 4
```

DESCRIPTION

Reads data from a stream that is opened by `open()` according to the conversion specifications provided in `format`, and then stores them in the arguments after `format`.

The results of the conversion that took place according to the conversion specifications provided in `format` are read for each element data of member functions 1 and 3, and as provided by the list of variable length arguments in `ap` for member functions 2 and 4.

The string buffer to be input-converted utilizes that returned by the `getline()` member function (§8.1.10). This then means that if `nchars` was not specified, the input conversion always gets performed on a per-line basis. The conversion characters that follow the `%` inside `format` and their features are provided in the table below:

Conversion character	Content of conversion	Type of argument
`hhd`	Converts inputs to signed decimal number	char type
`hd`	Converts inputs to signed decimal number	short type
`d`	Converts inputs to signed decimal number	int type
`ld`	Converts inputs to signed decimal number	long type
`lld`	Converts inputs to signed decimal number	long long type
`zd`	Converts inputs to signed decimal number	ssize_t type
`hhu`	Converts inputs to unsigned decimal number	unsigned char type
`hu`	Converts inputs to unsigned decimal number	unsigned short type
`u`	Converts inputs to unsigned decimal number	unsigned int type
`lu`	Converts inputs to unsigned decimal number	unsigned long type
`llu`	Converts inputs to unsigned decimal number	unsigned long long type
`zu`	Converts inputs to unsigned decimal number	size_t type
`hho`	Converts inputs to unsigned octal number	(unsigned) char type
`ho`	Converts inputs to unsigned octal number	(unsigned) short type
`o`	Converts inputs to unsigned octal number	(unsigned) int type
`lo`	Converts inputs to unsigned octal number	(unsigned) long type
`llo`	Converts inputs to unsigned octal number	(unsigned) long long type
`zo`	Converts inputs to unsigned octal number	size_t type and ssize_t type
`hhx`, `hhX`	unsigned hexadecimal number	(unsigned) char type
`hx`, `hX`	unsigned hexadecimal number	(unsigned) short type
`x`, `X`	unsigned hexadecimal number	(unsigned) int type
`lx`, `lX`	unsigned hexadecimal number	(unsigned) long type
`llx`, `llX`	unsigned hexadecimal number	(unsigned) long long type
`zx`, `zX`	unsigned hexadecimal number	size_t type and ssize_t type
`c`	Stores the string of the width specified in "maximum field width" (default value of 1) of an argument	char * type
`s`	Stores a string comprised of non-white space characters in an argument	char * type
`f`	Converts inputs to signed floating-point real number	float type
`lf`	Converts inputs to signed floating-point real number	double type
`e`, `E`	Converts inputs to signed floating-point real number	float type
`le`, `lE`	Converts inputs to signed floating-point real number	double type
`g`, `G`	Converts inputs to signed floating-point real number	float type
`lg`, `lG`	Converts inputs to signed floating-point real number	double type
`a`, `A`	Converts inputs to signed floating-point real number	float type
`la`, `lA`	Converts inputs to signed floating-point real number	double type
`p`	Converts inputs to void* pointer	void* type
`n`	Saves the number of characters consumed thus far from the input to the integer referenced by the int* pointer argument	int* type

The maximum field width can also be specified between `%` and a conversion character.

```
f_in.scanf("%__f",si_value);
           ↑
          [m]
```

Option	Meaning	Example
m (Input of number of digits)	Specifies maximum field width. Reads characters until this value is reached or a character that does not match is found.	`.scanf("%10d ...`

PARAMETER

[I]	`format`	Reading format specifications
[O]	`...`	Each element of data in which to write data
[O]	`ap`	List of variable length arguments in which to write data
[I]	`nchars`	Limit value for number of characters to be read

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Number of input elements successfully read and converted.
EOF (error)	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.
	:	If the byte array read was an invalid number.
	:	If the argument was too small or the `format` NULL.
	:	If the memory was insufficient.
	:	If, being converted to the integer type specified in `format`, the value was too large to be stored in the integer type.

EXCEPTION

If the system fails to secure the buffer used for reading the data.

EXAMPLE

The following code opens *file.txt* in *directory* in read mode, reads space-separated numerical values according to the format, and then stores them in `i_YY`,`i_MM` and `i_DD`. The stored values are then written to standard output according to the specified format. Please note that this example assumes that *directory/file.txt* has a year, month and date described in it in the format of YYYY␣MM␣DD.

```
stdstreamio f_in;
int i_YY = 0, i_MM = 0, i_DD = 0;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

if (f_in.scanf("%d %d %d", &i_YY, &i_MM, &i_DD) < 0) {
    Error handling
}

printf("%04d / %02d / %02d\n", i_YY, i_MM, i_DD);

f_in.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

When specifying "This problem can be avoided by specifying a maximum field width and then reading the string as follows. If you do not wish for any characters or newline characters that are not read to remain in the buffer, skip the strings up to the newline character in the manner shown below, and then skip the newline character.

```
stdstreamio f_in;
char c_buf[20];

f_in.scanf("%19s%*[^\n]",c_buf);
f_in.getchr();
```

8.1.12 putchr()

NAME

putchr() — Output of characters

SYNOPSIS

```
int putchr( int c );
```

DESCRIPTION

`putchr()` writes character `c` to stream opened by `open()`.

PARAMETER

[I] `c` Character to be written

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Normal termination, with the written unsigned char type character being cast to the int type.
EOF (error)	:	If the open mode for the input/output stream is inconsistent.
	:	If the system fails to write a stream for any other operating environment reason than described above. ⟦For example, a stream referenced by a file descriptor not having enough space.⟧

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

8.1.13 putstr()

NAME

putstr() — Output of strings

SYNOPSIS

```
int putstr( const char *s );
```

DESCRIPTION

`putstr()` writes string `s` to stream opened by `open()`.

PARAMETER

[I] `s` String to be written

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a string is successfully written. (`putstr()`)
EOF (error)	:	If the open mode for the input/output stream is inconsistent.
	:	If the system fails to write a stream for any other operating environment reason than described above. ⟦For example, a stream referenced by a file descriptor not having enough space.⟧

EXAMPLE

The following code opens *file.txt* in *directory* in write mode, and then writes the string `c_sentence` to it:

```
stdstreamio f_out;
const char *c_sentence = "This is an example code for the USER";

if (f_out.openf("w", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

if (f_out.putstr(c_sentence) < 0) {
    Error handling
}

f_out.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

8.1.14 printf(), vprintf()

NAME

printf(), vprintf() — Function that convert data to the set format

SYNOPSIS

```
int printf( const char *format, ... );
int vprintf( const char *format, va_list ap );
```

DESCRIPTION

Writes the content stored in arguments after `format` to a stream opened by `open()` according to the conversion specifications provided in `format`.

`printf()` converts each element of data, while `vprintf()`converts the list of variable length arguments in `ap`, depending on the conversion specifications provided in `format`.

The conversion characters that follow `%` inside `format` and their features are shown in the table below. Please note that if you want to output `%` itself you must type `%%`.

Conversion character	Content of conversion	Type of argument
`hhd`	Converts arguments to signed decimal number	char type
`hd`	Converts arguments to signed decimal number	short type
`d`	Converts arguments to signed decimal number	int type
`ld`	Converts arguments to signed decimal number	long type
`lld`	Converts arguments to signed decimal number	long long type
`zd`	Converts arguments to signed decimal number	ssize_t type
`hhu`	Converts arguments to unsigned decimal number	unsigned char type
`hu`	Converts arguments to unsigned decimal number	unsigned short type
`u`	Converts arguments to unsigned decimal number	unsigned int type
`lu`	Converts arguments to unsigned decimal number	unsigned long type
`llu`	Converts arguments to unsigned decimal number	unsigned long long type
`zu`	Converts arguments to unsigned decimal number	size_t type
`hho`	Converts arguments to unsigned octal number	(unsigned) char type
`ho`	Converts arguments to unsigned octal number	(unsigned) short type
`o`	Converts arguments to unsigned octal number	(unsigned) int type
`lo`	Converts arguments to unsigned octal number	(unsigned) long type
`llo`	Converts arguments to unsigned octal number	(unsigned) long long type
`zo`	Converts arguments to unsigned octal number	size_t type and ssize_t type
`hhx`, `hhX`	Converts arguments to unsigned hexadecimal number	(unsigned) char type
`hx`, `hX`	Converts arguments to unsigned hexadecimal number	(unsigned) short type
`x`, `X`	Converts arguments to unsigned hexadecimal number	(unsigned) int type
`lx`, `lX`	Converts arguments to unsigned hexadecimal number	(unsigned) long type
`llx`, `llX`	Converts arguments to unsigned hexadecimal number	(unsigned) long long type
`zx`, `zX`	Converts arguments to unsigned hexadecimal number	size_t type and ssize_t type
`c`	Outputs arguments as single character	int type
`s`	Outputs the string referenced by an argument. Characters up to the one before the null character or as many characters as the maximum number of characters indicated are output	const char * type
`f`	Receives arguments as float or double, and converts it to a decimal notation in the style [-]ddd.ddd	Floating-point type
`e`, `E`	Receives arguments as float or double, and converts it to a decimal notation in the style [-]d.ddddde[±]dd	Floating-point type
`g`, `G`	Uses the conversion with the smaller number of characters in conversion with `%e` or `%f`	Floating-point type
`a`, `A`	Receives arguments as float or double, and converts it to a hexadecimal notation in the style [-]0xh.hhhhhp[±]dd	Floating-point type
`p`	Receives arguments as the void* pointer, and outputs it as a hexadecimal number	void* type
`n`	Saves the number of characters written thus far to the integer referenced by the int* pointer argument	int* type

More detailed formatting specifications can also be made using the option specifiers between `%` and the conversion character as shown below:

```
sio.printf("%__f...\n",x);
```

$$\overbrace{[-][+][[0]m][.][n]}$$

Option	Meaning	Example
− (Minus sign)	Outputs a converted argument left-aligned. 1234_____	`.printf("%-6d ...`
+	Always prefixes a signed, converted numerical value with a sign (+ or −).	`.printf("%+6d ...`
m (Input of number of digits)	Specifies a minimum field width of m characters. If the converted value has fewer characters than specified, it will be padded with spaces on the left. Adding 0 results in zero padding. _____1234	`.printf("%10d ...` `.printf("%010d ...`
. (Period)	Separates a minimum filed width and the number of characters or the number of digits after the decimal point.	`.printf("%10.5f ...`

n (Input of number of digits)	If `f` is specified, number of digits after the decimal point.
	If `e`, `E`, `g` or `G` is specified, the accuracy.
	For strings, the width they require (including the beginning of the characters)
		`.printf("%10.5f ...`
		`.printf("%.15g ...`
		`.printf("%10.5s ...`

PARAMETER

[I] `format` Writing format specifications
[I] `...` Each element of the data to be written
[I] `ap` List of variable length arguments to be written

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value : Number of characters written.
Negative value (error) : If the system failed to write the data for an operating environment reason.

EXCEPTION

If the system failed to secure the buffer used for writing the data.

EXAMPLE

The following code writes the string `c_sentence` to a stream (standard output) in the

```
stdstreamio s_io;
const char *c_sentence = "This is an example code for the USER";
```

```
if (s_io.printf("%s\n", c_sentence) < 0) {
    Error handling
}
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

8.1.15 flush()

NAME

flush() — Forcefully outputs content of a stream.

SYNOPSIS

```
int flush();
```

DESCRIPTION

`flush()` forcefully writes all the data stored in the buffer inside a stream opened by `open()`.

RETURN VALUE

0	:	Normal termination
EOF (error)	:	If the stream is not opened.
	:	If the open mode for the input/output stream is inconsistent.
	:	If the system failed to forcefully output the data for any other operating environment reason than described above. ⟦For example, a stream referenced by a file descriptor not having enough space.⟧

EXAMPLE

The following code writes the string `c_sentence` to standard output stream in the

```
stdstreamio s_io;
const char *c_sentence = "This is an example code for the USER";

if (s_io.printf("%s\n", c_sentence) < 0) {
    Error handling
}

if (s_io.flush() < 0) {
    Error handling
}
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

8.1.16 eof(), error(), reseterr()

NAME

eof(), error(), reseterr() — Check and reset stream

SYNOPSIS

```
int eof();
int error();
cstreamio &reseterr();
```

DESCRIPTION

`eof()` tests the end-of-file indicator for the stream opened by `open()`, returning non-zero if it is set.

`error()` tests the error indicator for the stream opened by `open()`, returning non-zero if it is set.

`reseterr()` clears the end-of-file and error indicators for the stream opened by `open()`, and returns the object.

The corresponding functions in libc is `feof()`, `ferror()` and `clearerr()`, respectively.

WARNING

The cstreamio class is an abstract class and hence cannot be directly used. It must be used as a member function for the classes shown in Table 5.

8.1.17 seek(), rewind()

NAME

seek(), rewind() — Changes the position of streams

SYNOPSIS

```
int seek( long offset, int whence );
int rewind();
```

DESCRIPTION

`seek()` sets the stream position indicator on a per-byte basis of a stream opened by `open()`. The position to be set can be acquired by adding `offset` bytes to the position specified by `whence`.

`rewind()` sets a stream position indicator to the beginning of a file of a stream opened by `open()`.

PARAMETER

[I]	`offset`	Offset for stream position indicator
[I]	`whence`	Standard position for stream position indicator (`SEEK_SET` : Beginning of stream, `SEEK_CUR` : Present position indicator, `SEEK_END` : End of stream)

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the stream is not seekable.
	:	If the `whence` argument is inappropriate.
	:	If the system failed to process data for any other operating environment reason than described above. ⟦For example, when there is insufficient memory for the kernel.⟧

EXAMPLE

The following code opens *file.txt* in *directory* in read mode and changes the position of read operation to 40 bytes from the beginning of the file.

```
stdstreamio f_in;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

if (f_in.seek(40, SEEK_SET) < 0) {
    Error handling
}

f_in.close();
```

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.18 tell()

NAME

tell() — Current value of the stream position indicator

SYNOPSIS

`long tell();`

DESCRIPTION

`tell()` returns the current value of the stream position indicator for the stream opened by `open()`.

RETURN VALUE

Non-negative value	:	Present offset
Negative value (error)	:	If the stream is not seekable.
	:	If the system failed to process data for any other operating environment reason than described above. ⟦For example, when there is insufficient memory for the kernel.⟧

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.1.19 is_seekable()

NAME

is_seekable() — test for seekable stream or not

SYNOPSIS

`bool is_seekable();`

DESCRIPTION

`is_seekable()` returns `true` when disk seek is possible on the stream opened by `open()`.

RETURN VALUE

true	:	Disk seek is possible
false	:	Disk seek is impossible

WARNING

The cstreamio class is an abstract class and hence cannot be directly used by users. It must be used as a member function for the classes shown in Table 5.

8.2 The STDSTREAMIO class

Like `printf()` and `fopen()` in libc, the stdstreamio class is used to handle standard input/output, standard error output, and normal file input/output. As it also inherits cstreamio basically all the member functions in §8.1 are available for use in the class, with only
`open( const char *mode, cstreamio &sref )` not being usable. The class can be used without having to use `open()` and `close()` for standard input/output and standard error output.

If you use the stdstreamio class you must add "`#include <sli/stdstreamio.h>`" to the code. If you need to declare a namespace (§4.1), you must also add "`using namespace sli;`" to the code. Table 8 lists the member functions. The "Corresponding function in libc" in Table 8 shows the corresponding functions in libc with the same feature as each of the member functions.

	The stdstreamio class	Feature	Corresponding function in libc
§8.2.2	`open()`, `openf()`, `vopenf()`	Opens a stream	`fopen()`
§8.1.2	`close()`	Closes a stream	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Seek forward (if possible) or read n bytes to skip data stream	—
§8.1.7	`wskip()`	Seek forward (if possible) or write n bytes of blank data	—
§8.1.8	`getchr()`	Input of character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of line	—
§8.1.11	`scanf()`	Converts inputs using format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs content of buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.2.3	`eprintf(...)`	Outputs to standard error output	`fprintf(stderr, ...)`
§8.2.4	`eflush()`	Flush for standard error output	`fflush(stderr)`
§8.2.5	`seek()`	Change position of streams	`fseek()`
§8.2.5	`rewind()`	Changes position of streams to the beginning	`rewind()`
§8.2.6	`tell()`	Value of stream position indicator	`ftell()`
§8.1.19	`is_seekable()`	Test for seekable stream or not	-

Table 8: List of the member functions available with stdstreamio class.

How to use the member functions that have been redefined and added with the stdstreamio class is described below.

8.2.1 How to create an object

With the stdstreamio class if you do not provide an argument when creating an object, the object gets output to standard output when generated, as revealed in the example below:

```
#include <sli/stdstreamio.h>
using namespace sli;

int main()
{
    stdstreamio sio;
    sio.printf("Hello\n");
```

Creating an object with a flag, as revealed below, allows the location to be switched to where the object is output when generated to standard error output.

```
    stdstreamio eout(true);
    eout.printf("This is STDERR\n");
```

8.2.2 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Opens stream

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, const char *path );  ..... 3
int openf( const char *mode, const char *path_fmt, ... );  ..... 4
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 5
```

DESCRIPTION

Opens the file indicated by `path` or `path_fmt`, or attaches the descriptor specified by `fd` to a stream. If `path` or `path_fmt` is `NULL`, the standard input or standard output is used.

With the `mode` for member functions 1 and 2, `"r"` is specified when reading data, and `"w"` when writing data. In addition, either `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"` or `"a+"` can be specified for member functions 3 to 5. Details on the `mode` that can be specified are provided in the table below:

mode	Processing	If the file does not exist
`"r"`	Only reading	Abnormal termination
`"r+"`	Reading and writing	Abnormal termination
`"w"`	Only writing	Creation of new file
`"w+"`	Reading and writing	Creation of new file
`"a"`	Only appending	Creation of new file
`"a+"`	Reading and appending	Creation of new file

For more details on member functions 4 and 5 refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	File opening mode
[I]	`fd`	File descriptor
[I]	`path`	File name
[I]	`path_fmt`	File name format specifications
[I]	`...`	Each element of data for file name
[I]	`ap`	List of variable length arguments for file name

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination.
Negative value (error)	:	If the system failed to open the stream because the file does not exist etc.
	:	If the system failed to open the stream because the specified `mode` is inappropriate etc.
	:	If the system failed to open the stream because it is not allowed to access the stream by the specified mode etc.
	:	If the string indicates the path of `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the member functions shown in this section.

EXCEPTION

If the system fails to copy a file descriptor for the standard input/output and standard error output.

EXAMPLE

The following code opens in read mode *file.txt* in *directory*.

```
stdstreamio f_in;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

f_in.close();
```

8.2.3 eprintf(), veprintf()

NAME

eprintf(), veprintf() — Converts data to the set format, and then outputs it to standard error

SYNOPSIS

```
int eprintf( const char *format, ... );
int veprintf( const char *format, va_list ap );
```

DESCRIPTION

Writes the content stored in arguments after `format` to the standard error output stream according to the conversion specifications provided in `format`.

`eprintf()` converts each element of data, while `veprintf()` converts the list of variable length arguments in `ap`, depending on the conversion specifications provided in `format`.

The conversion characters that follow `%` inside `format` and their features are provided in the table in §8.1.14. Please note that if you wish to output `%` you must type `%%`.

PARAMETER

[I]	`format`	Writing format specifications
[I]	`...`	Each element of data to be written
[I]	`ap`	List of variable length arguments to be written

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Number of characters written
Negative value (error)	:	If the system failed to write data for an operating environment reason.

EXAMPLE

The following code writes the string `c_error` to the standard error output in the %s format:

```
stdstreamio s_io;
const char *c_error = "This example code doesn't work correctly";

if (s_io.eprintf("%s\n", c_error) < 0) {
    Error handling
}
```

8.2.4 eflush()

NAME

eflush() — Forcefully outputs the content of standard error output

SYNOPSIS

```
int eflush();
```

DESCRIPTION

`eflush()` forcefully writes all the data stored in the buffer inside the standard error output stream.

RETURN VALUE

0	:	Normal termination.
EOF (error)	:	If the system failed to forcefully output data for an operating environment reason.

EXAMPLE

The following code writes the string `c_error` to the standard error output stream in the %s format, and then forcefully outputs the buffer.

```
stdstreamio s_io;
const char *c_error = "This example code doesn't work correctly";

if (s_io.eprintf("%s\n", c_error) < 0) {
    Error handling
}

if (s_io.eflush() < 0) {
    Error handling
}
```

8.2.5 seek(), rewind()

NAME

seek(), rewind() — Changes the position of streams

SYNOPSIS

```
int seek( long offset, int whence );
int rewind();
```

DESCRIPTION

`seek()` sets the stream position indicator on a per-byte basis of a stream opened by `open()`. The position to be set can be acquired by adding `offset` bytes to the position specified by `whence`.

`rewind()` sets a stream position indicator to the beginning of a file of a stream opened by `open()`.

PARAMETER

[I]	`offset`	Offset for stream position indicator
[I]	`whence`	Standard position for stream position indicator (`SEEK_SET` : Beginning of stream, `SEEK_CUR` : Present position indicator, `SEEK_END` : End of stream)

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the stream is not seekable.
	:	If the `whence` argument is inappropriate.
	:	If the system failed to process data for any other operating environment reason than described above. ⟦For example, when there is insufficient memory for the kernel.⟧

EXAMPLE

The following code opens *file.txt* in *directory* in read mode and changes the position of read operation to 40 bytes from the beginning of the file.

```
stdstreamio f_in;

if (f_in.openf("r", "%s/%s", "directory", "file.txt") < 0) {
    Error handling
}

if (f_in.seek(40, SEEK_SET) < 0) {
    Error handling
}

f_in.close();
```

8.2.6 tell()

NAME

tell() — Current value of the stream position indicator

SYNOPSIS

```
long tell();
```

DESCRIPTION

`tell()` returns the current value of the stream position indicator for the stream opened by `open()`.

RETURN VALUE

Non-negative value	:	Present offset
Negative value (error)	:	If the stream is not seekable.
	:	If the system failed to process data for any other operating environment reason than described above. ⟦For example, when there is insufficient memory for the kernel.⟧

8.2.7 content_length()

NAME

content_length() — Calls for the length of streams

SYNOPSIS

```
long long content_length() const;
```

DESCRIPTION

Returns the byte length of a stream opened by `open()`.

RETURN VALUE

Non-negative value	:	Byte length of stream
Negative value (error)	:	If length of stream cannot be obtained.

8.3 GZSTREAMIO class

The gzstreamio class is used to handle standard input/output and normal file input/output while also performing gzip compression/expansion. It inherits cstreamio and hence all the member functions detailed in §8.1 are available for use with the class. `open()`, `close()`must always be used with the gzstreamio class.

If you with to use the gzstreamio class, you must add "`#include <sli/gzstreamio.h>`" to the code. If you need to declare a namespace (§4.1), you must also add "`using namespace sli;`" to the code.

Table 9 lists the member functions. The "Corresponding function in libc" column in Table 9 shows the corresponding functions in libc with the same feature as each of the member functions.

	The gzstreamio class	Feature	Corresponding function in libc
§8.3.1	`open()`, `openf()`, `vopenf()`	Opens streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of line	—
§8.1.11	`scanf()`	Converts inputs using format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs content of buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.3.2	`sync()`	Forcefully outputs content of buffer	`fflush()`

Table 9: List of member functions available for use with gzstreamio class.

How to use the member functions redefined and added in the gzstreamio class is described below.

8.3.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Opens streams

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, cstreamio &sref );  ..... 3
int open( const char *mode, const char *path );  ..... 4
int openf( const char *mode, const char *path_fmt, ... );  ..... 5
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 6
```

DESCRIPTION

Opens the gzip format file indicated by `path` or `path_fmt` or attaches a file descriptor in the gzip format specified by `fd` or the object for an inherited class of the cstreamio class specified by `sref` to a gzip format stream. If `path` or `path_fmt` is `NULL`, the standard input or standard output is used.

With the `mode` for the member functions of 1, 2, 4, 5 or 6, `"rb"` is specified when reading data and `"wb"` when writing data. Specifying `"b"` in this manner enables the mode to be forcefully switched to binary mode. It is strongly recommended that users specify `"b"` when binary files will be handled as otherwise problems could occur such as damage to image files. With the `mode` for the member function of 3, `"r"` must be specified when reading data and `"w"` when writing data.

When writing data, you can specify the level and method of compression using the `mode`. In this case the mode needs to be be specified in the
`[fopen mode][level of compression [method of compression]]` format, as in `"wb6f"`. The levels of compression and methods are provided in the table below. The default value for the compression level, which is the average speed and compression ratio, is 6.

Character specified	Level of compression
`0`	No compression
`1`	The speed of processing is of primary concern
⋮	
`9`	The efficiency of compression is of primary concern

Character specified	Method of compression
`f`	Filter data
`h`	Compression using only the Huffman method (For data with many of the same binary sets such as word)
`R`	Run length encoding (For data with sequential binaries such as images)

The member function 1 is used in cases such as the standard input/output needing to be redirected to a gzip-formatted file (Refer to EXAMPLE).

The member function 3 is used in cases such as when an object for an inherited class of the cstreamio class has had a stream already opened by it, or the stream gets changed during the course of the stream to a gzip-compressed stream (Refer to EXAMPLE-2 in §8.1.1).

For more details on the arguments used with `path_fmt` and thereafter for member functions 5 and 6 refer to the descriptions in §8.1.1.

PARAMETER

[I]	`mode`	File opening mode
[I]	`fd`	File descriptor
[I]	`sref`	Object for an inherited class of the cstreamio class
[I]	`path`	Name of file
[I]	`path_fmt`	Specifications for file name format
[I]	`...`	Each element of data of a file name
[I]	`ap`	All the elements of data of a file name

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the system failed to open the stream because the file does not exist etc.
	:	If the system failed to open the stream because the `mode` specified was inappropriate, etc.
	:	If the system failed to open the stream because the relationship between the `mode` specified and `fd` was incorrect etc (Member function 2).
	:	If the system failed to open the stream because cannot access the stream in the mode specified.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the member functions detailed in this section.
	:	If the system failed to open the stream for the reason that a compression method that is not supported is specified, etc.
	:	When the end of the stream is reached unexpectedly.

EXCEPTION

If the system fails to copy a file descriptor for the standard input/output (Member functions 1 and 4).
If the system fails to secure a temporary buffer for the compression algorithm (Member function 3).
If the system fails to initialize the compression algorithm (Member function 3).
If the system fails to secure output buffer (Member function 3).

EXAMPLE

The following code redirects the standard input to the gzip-formatted *file.txt.gz*, opens it in read mode, and then print the content to standard output. If *gzstreamio_open* is the name of the executable file for this code, run the following command:

```
$ ./gzstreamio_open < file.txt.gz
```

```
#include <sli/gzstreamio.h>
using namespace sli;

int main()
{
    gzstreamio s_io;
    const char *line_ptr;

    if (s_io.open("rb") < 0) {
         Error handling
    }

    while ((line_ptr = s_io.getline()) != NULL) {
        printf("%s", line_ptr);
    }

    s_io.close();

    return 0;

}
```

WARNING

The functions can also open files that are not gzip-formatted, but be aware that writing and reading those files may not allow the intended processing to be performed.

8.3.2 sync()

NAME

sync() — Forcefully outputs the content, and adjusts it so that it terminates at a byte boundary.

SYNOPSIS

```
int sync();
```

DESCRIPTION

With `flush()` (§8.1.15), the outputs does not necessarily get terminated at a byte boundary. `sync()` calls `flush()`, and then adjusts the output so that it terminates at a byte boundary. Use of this member function therefore often reduces the compression ratio. Most applications do not require the `sync()` member function.

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the system fails to gzip-compress any data.

EXAMPLE

The following code writes binary data, gzip-compressed from the string `c_sentence`, to *file.txt.gz* in *directory*.

```
gzstreamio f_out;
const char *c_sentence = "This is an example code for the USER";

if (f_out.openf("wb", "%s/%s", "directory", "file.txt.gz") < 0) {
    Error handling
}

if (f_out.printf("%s", c_sentence) < 0) {
    Error handling
}

if (f_out.sync() < 0) {
    Error handling
}

f_out.close();
```

8.4 The BZSTREAMIO class

The bzstreamio class is used to handle the standard input/output and normal file input/output while also performing bzip2 compression/expansion. It inherits cstreamio and hence all the member functions in §8.1 are available for use with the class. With the bzstreamio class, `open()` and `close()` must always be used.

If you wish to use the bzstreamio class you must add "`#include <sli/bzstreamio.h>`" to the code. If you need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

Table 10 lists the member functions. The "Corresponding function in libc" column in Table 10 provides the functions in libc with the same feature as each of the member functions.

bzip2 is inferior to gzip in terms of processing speed, but has a data compression algorithm that supports a higher ratio of compression.

	The bzstreamio class	Feature	Corresponding function in libc
§8.4.1	`open()`, `openf()`, `vopenf()`	Open streams	`fopen()`
§8.1.2	`close()`	Close streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of line	—
§8.1.11	`scanf()`	Converts inputs with a format and assigns them to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.

Table 10: bzstreamio List of member functions available for use with the bzstreamio class.

How to use the member functions redefined and added in the bzstreamio class is described below.

8.4.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Open streams

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, cstreamio &sref );  ..... 3
int open( const char *mode, const char *path );  ..... 4
int openf( const char *mode, const char *path_fmt, ... );  ..... 5
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 6
```

DESCRIPTION

Opens a bzip2-formatted file indicated by `path` or `path_fmt`, or attaches a bzip2-formatted file descriptor specified by `fd` or the object for an inherited class of the cstreamio class specified by `sref` to a bzip2-formatted stream. If `path` or `path_fmt` is `NULL`, the standard input or standard output is used.

With the `mode` for the member functions of 1, 2, 4, 5 or 6, `"rb"` is specified when reading data and `"wb"` when writing data. Specifying `"b"` in this manner enables the mode to be forcefully switched to binary mode. It is strongly recommended that users specify `"b"` when handling binary files as otherwise problems could occur such as damage to image files. With the `mode` for the member function of 3, `"r"` must be specified when reading data and `"w"` when writing data.

Member function 1 is used in cases such as the standard input/output being redirected to a bzip2-formatted file (Refer to EXAMPLE-1).

The member function 3 is used when an object for an inherited class of the cstreamio class has had a stream already opened by it, or the stream gets changed during the course of the stream to a bzip2-compressed stream (Refer to EXAMPLE-2 in §8.1.1).

For more details on the arguments for `path_fmt` and thereafter for member functions 5 and 6, refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	File opening mode
[I]	`fd`	File descriptor
[I]	`sref`	Object for inherited class of cstreamio class
[I]	`path`	Name of file
[I]	`path_fmt`	File name format specifications
[I]	`...`	Each element of data of a file name
[I]	`ap`	All the elements of data of a file name

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the system failed to open the stream because the file does not exist etc.
	:	If the system failed to open the stream because the `mode` specified was inappropriate etc.
	:	If the system failed to open the stream because the relationship between the `mode` specified and `fd` was incorrect etc (Member function 2).
	:	If the system failed to open the stream because it cannot access the stream in the mode specified etc.
	:	If the string indicating `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the member functions detailed in this section.

EXCEPTION

If the system failed to copy a file descriptor for the standard input/output (Member functions 1 and 4).
If the system failed to secure a temporary buffer for the compression algorithm (Member function 3).
If the system failed to initialize the compression algorithm (Member function 3).
If the system failed to secure the output buffer (Member function 3).

EXAMPLE-1

The following code redirects the standard input to the bzip2-formatted *file.txt.bz2*, opens it in read mode, and then print the content to standard output. If *bzstreamio_open* is the name of the executable file for this code, run the following command:

```
$ ./bzstreamio_open < file.txt.bz2
```

```
#include <sli/bzstreamio.h>
using namespace sli;

int main()
{
    bzstreamio s_io;
    const char *line_ptr;

    if (s_io.open("rb") < 0) {
         Error handling
    }

    while ((line_ptr = s_io.getline()) != NULL) {
        printf("%s", line_ptr);
    }

    s_io.close();

    return 0;
}
```

EXAMPLE-2

The following code opens the bzip2-formatted *file.txt.bz2* in *directory* in read and binary mode, and then prints the content to standard output.

```
bzstreamio f_in;
const char *line_ptr;

if (f_in.open("rb", "directory/file.txt.bz2") < 0) {
    Error handling
}

while ((line_ptr = f_in.getline()) != NULL) {
    printf("%s", line_ptr);
}

f_in.close();
```

WARNING

The functions can also open files that are not bzip2-formatted but be aware that writing and reading them may not allow the intended processing to be performed.

8.5 The HTTPSTREAMIO class

The httpstreamio class uses the GET method of an http server to perform stream inputs from that http server. It can also input streams while performing gzip expansion or bzip2 expansion.

The class inherits cstreamio, but some of the member functions in §8.1 are not available. For the member functions available with the httpstreamio class refer to the list of member functions in Table 11. With the httpstreamio class, `open()` and `close()` must always be used. With the `open()` member function, `"r"` or `"r%"` is specified as the `mode`, and a URL that begins with `http://` is specified as the `path`. Connections through a proxy server are not supported.

If you wits to use the httpstreamio class, you must add "`#include <sli/httpstreamio.h>`" to the code. If you need to declare a namespace (§4.1), you must also add "`using namespace sli;`" to the code.

The "Corresponding function in libc" column in Table 11 provides the corresponding functions in libc with the same feature as each of the member functions.

	The httpstreamio class	Feature	Corresponding function in libc
§8.5.1	`open()`, `openf()`, `vopenf()`	Open streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.8	`getchr()`	Input of character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of line	—
§8.1.11	`scanf()`	Converts inputs with a format and assigns to an argument	`fscanf()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.5.2	`content_length()`	Calls for the length of streams	—
§8.5.3	`user_agent().assign()`	Sets user agent	—

Table 11: List of member functions available for use with the httpstreamio class.

How to use the member functions redefined and added in the httpstreamio class is described below.

8.5.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Open streams

SYNOPSIS

```
int open( const char *mode, const char *path );
int openf( const char *mode, const char *path_fmt, ... );
int vopenf( const char *mode, const char *path_fmt, va_list ap );
```

DESCRIPTION

Opens a URL indicated by `path` or `path_fmt`. `"r"` or `"r%"` is specified as the `mode`. If `"r%"` is specified as the `mode`, gzip- or bzip2-compressed streams are expanded when read if necessary. The header information (MIME) that a server returns and the suffix ("`.gz`" or "`.bz2`") of a file name for `path` or `path_fmt` determine whether gzip or bzip2 is used.

For more details on the arguments for `path_fmt` and thereafter for `openf()` and `vopenf()`, refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	URL opening mode
[I]	`path`	Path for URL
[I]	`path_fmt`	URL format specifications
[I]	`...`	Each element of data of URL
[I]	`ap`	All the elements of data of URL

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the system failed to open the stream because the URL specified was inappropriate etc.
	:	If the system failed to open the stream because the `mode` was not specified etc.
	:	If the system failed to open the stream because the `mode` specified was inappropriate etc.
	:	If the system failed to open the stream because it cannot access the stream in the mode specified.
	:	If the string indicating `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the member functions detailed in this section.
	:	If the system failed to open the gzip- or bzip2-formatted stream.

EXCEPTION

If the system fails to allocate enough memory.

EXAMPLE

The following code opens the URL of *http://www.jaxa.jp/* in read mode, and then prints the content of the linked source code (html) to standard output.

```
httpstreamio net_in;
const char *line_ptr;

if (net_in.open("r", "http://www.jaxa.jp/") < 0) {
    Error handling
}

while ((line_ptr = net_in.getline()) != NULL) {
    printf("%s", line_ptr);
}

net_in.close();
```

8.5.2 content_length()

NAME

content_length() — Calls for the length of streams

SYNOPSIS

```
long long content_length() const;
```

DESCRIPTION

Returns the byte length of a stream opened by `open()`. With compressed streams, it returns the byte length of a compressed stream.

RETURN VALUE

Non-negative value	:	Byte length of stream
Negative value (error)	:	If Content-Length information does not exist in the MIME header.

EXAMPLE

The following code opens the URL of *http://www.jaxa.jp/* in read mode, and then prints the byte length of the stream to standard output.

```
httpstreamio net_in;
long long l_ret;

if (net_in.open("r", "http://www.jaxa.jp/") < 0) {
    Error handling
}

if ((l_ret = net_in.content_length()) < 0) {
    printf("No information about stream byte size\n");
}
else {
    printf("Stream Byte Size = %lld \n", l_ret);
}

net_in.close();
```

8.5.3 user_agent().assign()

NAME

user_agent().assign() — Sets user agent

SYNOPSIS

```
tstring &user_agent().assign( const char *uagent );
tstring &user_agent().assignf( const char *uagent_fmt, ... );
```

DESCRIPTION

Sets the user agent to be transmitted when connecting to a Web server. If a user agent is not set, "hostname SLLIB-x.x::httpstreamio" will be transmitted.

8.6 The FTPSTREAMIO class

The ftpstreamio class connects to an ftp server and performs the stream input from that ftp server or stream output to the ftp server in passive mode. The class can also input/output streams while performing gzip expansion/compression or bzip2 expansion/compression.

As the class inherits cstreamio basically all the member functions in §8.1 are available for the class, with only `open( const char *mode, cstreamio &sref )` not being usable. With the ftpstreamio class `open()` and `close()`must always be used. With the `open()` member functions, `"r"`, `"r%"`, `"w"` or `"w%"` is specified as the `mode`, and a URL that begins with `ftp://` is specified as the `path`. Connections through a proxy server are not supported.

If you wish to use the ftpstreamio class you must add "`#include <sli/ftpstreamio.h>`" to the code. If you need to declare a namespace (§4.1), you must also add "`using namespace sli;`" to the code.

Table 12 lists the member functions. The "Corresponding function in libc" column in Table 12 provides the corresponding functions in libc with the same feature as each of the member functions.

	The ftpstreamio class	Feature	Corresponding function in libc
§8.6.1	`open()`, `openf()`, `vopenf()`	Open streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of line	—
§8.1.11	`scanf()`	Converts inputs with format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.6.2	`content_length()`	Calls for the length of streams	—
§8.6.3	`username().assign()`	Sets the name of FTP user	—
§8.6.4	`password().assign()`	Sets the password for FTP user	—

Table 12: List of the member functions available for use with the ftpstreamio class.

How to use the member functions redefined and added in the httpstreamio class is described below.

8.6.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Open streams

SYNOPSIS

```
int open( const char *mode, const char *path );
```

```
int openf( const char *mode, const char *path_fmt, ... );
int vopenf( const char *mode, const char *path_fmt, va_list ap );
```

DESCRIPTION

Opens a URL indicated by `path` or `path_fmt`. `path` must always begin with `ftp://`. A user name and password can be included in `path` by making the appropriate setting in the `ftp://username:password@ftp.com/` ... format. If a user name and password were not included in `path`, they can also be set using `username().assign()` and `password().assign()` (§8.6.3 and 8.6.4). If a user name and password are not set, connections will be made to the ftp server anonymously.

`"r"`, `"r%"`, `"w"` or `"w%"` is specified as the `mode`. When `"r%"` is specified as the `mode`, gzip- or bzip2-compressed streams are expanded when read if necessary. When `"w%"` is specified as the `mode`, gzip or bzip2 is used to compresses a stream if necessary and then send it to the ftp site. The suffix ("`.gz`" or "`.bz2`") used for the file name with `path` determines whether gzip or bzip2 is used.

For more details on the arguments for `path_fmt` and thereafter for `openf()` and `vopenf()`, refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	URL opening mode
[I]	`path`	Path for URL
[I]	`path_fmt`	URL format specifications
[I]	`...`	Each element of data of a URL
[I]	`ap`	All the elements of data of a URL

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (error)	:	If the system failed to open the stream because the URL was not specified etc.
	:	If the system failed to open the stream because the URL specified was inappropriate etc.
	:	If the system failed to open the stream because the `mode` specified was inappropriate etc.
	:	If the string indicating `path_fmt` exceeds `PATH_MAX`.
	:	If the system failed to open the stream because the `mode` was not specified etc.
	:	If the system failed to open the stream because it cannot access the stream in the mode specified etc.
	:	If the stream has already been opened by any of the member functions detailed in this section.
	:	If the system failed to open the stream because the system failed to connect to an ftp server etc.
	:	If the system failed to open the gzip- or bzip2-formatted stream.

EXCEPTION

If the system failed to allocate enough memory.

EXAMPLE

The following code opens the URL of *ftp://ftp.boof.com/file.txt* in read mode, and then prints the content to standard output. Please note that the URL provided on the left does

not actually exist and hence execution the code will result in open() abnormally terminating. If an actual in use URL were used the code would operate normally.

```
ftpstreamio f_in;
const char *line_ptr;

if (f_in.open("r", "ftp://ftp.boof.com/file.txt") < 0) {
    Error handling
}

while ((line_ptr = f_in.getline()) != NULL) {
    printf("%s", line_ptr);
}

f_in.close();
```

8.6.2 content_length()

NAME

content_length() — Calls for the length of streams

SYNOPSIS

```
long long content_length() const;
```

DESCRIPTION

With streams opened with a read mode specification in `open()`, the byte length of a stream is returned. With compressed streams opened by `open()`, the byte length of a compressed stream is returned.

RETURN VALUE

Non-negative value	:	Byte length of a stream
Negative value (error)	:	If the stream is not opened.

EXAMPLE

The following code opens the URL of *ftp://ftp.boof.com/file.txt* in read mode, and then prints the byte length of the stream to standard output. Please note that the URL provided on the left does not actually exist and hence execution of the code will result in open() abnormally terminating. If an actual in use URL were to be used the code would operate normally.

```
ftpstreamio f_in;
long long l_ret;

if (f_in.open("r", "ftp://ftp.boof.com/file.txt") < 0) {
    Error handling
}

if ((l_ret = f_in.content_length()) < 0) {
    printf("No information about stream byte size\n");
}
else {
    printf("Stream Byte Size = %lld \n", l_ret);
}
```

```
f_in.close();
```

8.6.3 username().assign()

NAME

username().assign() — Sets the name for FTP user

SYNOPSIS

```
tstring &username().assign( const char *user );
```

DESCRIPTION

By default, the `open()`member function (§8.6.1) enables users to log on to an FTP server anonymously, but a user name can be set using this member function.

8.6.4 password().assign()

NAME

password().assign() — Sets the password for FTP user

SYNOPSIS

```
tstring &password().assign( const char *pass );
```

DESCRIPTION

Sets the password for an FTP user assigned by `username().assign()`.

8.7 The PIPESTREAMIO class

The pipestreamio class is used to execute files and then connect the process executed to input or output stream pipes. The class inherits cstreamio and therefore basically all the member functions in §8.1 except `open( const char *mode, cstreamio &sref )` are available for use. With the pipestreamio class `open()` and `close()` must always be used. With the `open()` member function, `"r"` or `"w"` is specified as the `mode`, while a command is specified as the `path`. The mode being `"r"` denotes input from the executed process, but when `"w"` denotes output to the executed process. `path` can include command options and symbols for a pipe or redirection (`|`,`<`,`>`). The `open()` member function is also available that enables commands to be specified using arguments for `char *const argv[]`, for example `execvp()`, from libc.

If you wish to use the pipestreamio class you must add "`#include <sli/pipestreamio.h>`" to the code. In addition, if you need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

The "Corresponding function in libc" column in Table 13 provides the corresponding functions in libc with the same feature as each of the member functions.

	pipestreamio class	Feature	Corresponding function in libc
§8.7.1	`open()`, `openf()`, `vopenf()`	Opens streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of single character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of single line	—
§8.1.11	`scanf()`	Converts input with a format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of single character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of a buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.

Table 13: List of member functions available for use with the pipestreamio class.

How to use the member functions redefined and added in the pipestreamio class is described below.

8.7.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Opens streams

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, const char *path );  ..... 3
```

```
int open( const char *mode, const char *const argv[] );  ..... 4
int openf( const char *mode, const char *path_fmt, ... );  ..... 5
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 6
```

DESCRIPTION

Member functions 3 to 6 are used to execute the command indicated by `path`, `argv` or `path_fmt`, and then connect the result using a pipe to open the stream. Pipes provide unidirectional inter-process communication channels, and have both "read" and "write" sides. Data written to the write side of a pipe can then be read from the read side of the pipe. The `mode` being `"r"` results in the specified command being executed, with the output from standard output of the command being connected to a pipe so that it can then be read by the reading member function of the object (Refer to EXAMPLES 1 and 2). The `mode` being `"w"` results in the specified command being executed, with the result being written by writing member functions of the object to a pipe so that it can then be read as the standard input for the command (Refer to EXAMPLE 3).

`path` or `path_fmt` being set results in it being executed in the "`/bin/sh -c` *command*" format, and hence `path` or `path_fmt` can include pipe and redirection symbols (`|`,`<`,`>`). If `path` or `path_fmt` is `NULL`, the standard input or standard output is used.

`argv[]` elements must be set in the order of "*the executable file's path name, argument 1, argument 2,* ... `NULL`". The end of `argv[]` must always be `NULL`.

Member functions 1 and 2 cannot be used to execute a command and connect the result with a pipe to open the stream. For more details on these member functions refer to the descriptions provided in §8.1.1.

For more details on the arguments used with `path_fmt` and later for member functions 5 and 6 refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	Stream opening mode
[I]	`fd`	File descriptor
[I]	`path`	A command
[I]	`path_fmt`	Command format specifications
[I]	`...`	Each element of data of a command
[I]	`ap`	All the elements of data of a command
[I]	`argv[]`	All the elements of data of a command

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (Error)	:	If the system failed to open a stream because a command was not specified etc.
	:	If the system failed to open a stream because the `mode` specified was inappropriate etc.
	:	If the system failed to open a stream because it cannot access the stream in the specified mode etc.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the other member functions described in this section.

EXCEPTION

If the system failed to generate a file descriptor for use in reading or writing data.
If the system failed to generate a process.

EXAMPLE-1

The following code formats the man page for the function fprintf, and then writes the content of the page to the output file opened in write mode:

```
stdstreamio  f_out;
pipestreamio p_in;
const char  *argv[4] = { "man", "fprintf", NULL };
const char  *line_ptr;

f_out.openf("w", "%s", "file.txt");

if (p_in.open("r", argv) < 0) {
    Error handling
}

while ((line_ptr = p_in.getline()) != NULL) {
    f_out.printf("%s",line_ptr);
}

p_in.close();
f_out.close();
```

EXAMPLE-2

The following code sorts the content of *file.txt* in *directory* using the second field as the key and on a per-line basis, and then prints the result to standard output:

```
pipestreamio p_in;
const char *line_ptr;

if (p_in.open("r", "cat directory/file.txt | sort -k 2") < 0) {
    Error handling
}

while ((line_ptr = p_in.getline()) != NULL) {
    printf("%s",line_ptr);
}

p_in.close();
```

EXAMPLE-3

The following code reads the content of *file.txt* in *directory* one line at a time. The content of each line is read and the line that matches the string *pattern* is written to standard output:

```
stdstreamio f_in;
pipestreamio p_out;
const char *line_ptr;

if (f_in.open("r", "directory/file.txt") < 0) {
    Error handling
}
```

```
if (p_out.open("w", "grep pattern") < 0) {
    Error handling
}

while ((line_ptr = f_in.getline()) != NULL) {
    p_out.printf("%s",line_ptr);
}

p_out.close();
f_in.close();
```

8.8 The DIGESTSTREAMIO class

The digeststreamio class is used to handle the stream input/output of the URL or file indicated by the `path` argument of the `open()` member function while performing gzip- or bzip2-compression/expansion if necessary. The `openp()` member function added in this class also additionally enables input from commands and output to commands using a pipe. The digeststreamio class is therefore an extremely versatile class of the inherited cstreamio classes. The header information (MIME) returned by a server and the suffix of the file name in `path` provided to `open()`can be used to determine whether any compression/expansion is necessary. The class inherits cstreamio and hence basically all the member functions in §8.1 are available for use, apart from `open( const char *mode, cstreamio &sref )`. With the digeststreamio class `open()` and `close()` must always be used. `"r"` or `"w"` must always be specified as the `mode` for the `open()`member function. Output to an http server is currently not supported, along with connections through a proxy server.

If you wish to use the digeststreamio class you must add "`#include <sli/digeststreamio.h>`" to the code. In addition, if you need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

Table 14 lists the member functions. The "Corresponding function in libc" column in Table14 provides the corresponding functions in libc with the same feature as each of the member functions.

	digeststreamio class	Feature	Corresponding function in libc
§8.8.1	`open()`, `openf()`, `vopenf()`	Opens streams	`fopen()`
§8.8.2	`openp()`, `openpf()`, `openpf()`	Opens streams (perl-like)	—
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Seek forward (if possible) or read n bytes to skip data stream	—
§8.1.7	`wskip()`	Seek forward (if possible) or write n bytes of blank data	—
§8.1.8	`getchr()`	Input of single character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of single line	—
§8.1.11	`scanf()`	Converts inputs with a format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of single character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of a buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.

Table 14: List of the member functions available for use with the digeststreamio class.

How to use the member functions redefined and added in the digeststreamio class is described below.

8.8.1 open(), openf(), vopenf()

NAME

open(), openf(), vopenf() — Opens streams

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, const char *path );  ..... 3
int openf( const char *mode, const char *path_fmt, ... );  ..... 4
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 5
```

DESCRIPTION

Opens the file or URL indicated by `path` or `path_fmt`. With URLs a string that begins with `file://`, `http://` or `ftp://` can be specified. If `path` or `path_fmt` is `NULL` the standard input or standard output is used.

The `mode` being `"r"` performs input and when `"w"` output. Member functions 1 and 2 cannot execute commands and connect the result using a pipe to open the stream. For more details on these member functions refer to the descriptions provided in §8.1.1.

For more details on the argument for `path_fmt` and later for member functions 4 and 5 refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	File or URL opening mode
[I]	`fd`	File descriptor
[I]	`path`	File or URL
[I]	`path_fmt`	File or URL format specifications
[I]	`...`	Each element of data of a file or URL
[I]	`ap`	All the elements of data of a file or URL

([I] : input, [O] : output)

RETURN VALUE

0	:	Normal termination
Negative value (Error)	:	If the system failed to open a stream because the `mode` specified was inappropriate etc.
	:	If the system failed to open a stream because the relationship between the `mode` specified and `fd` was incorrect etc (Member function 2).
	:	If the system failed to open a stream because it cannot access the stream in the specified mode etc.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the other member functions described in this section.
	:	If the system failed to open the file or URL.

EXCEPTION

If the system failed to generate an object.

EXAMPLE

The following code first opens the URL of *http://www.jaxa.jp/* in read mode and then writes the binary data from the gzip-compressed content of that stream to *file.gz* in *directory*.

```
digeststreamio net_in;
```

```
digeststreamio f_out;
const char *line_ptr;

if (net_in.open("r", "http://www.jaxa.jp/") < 0) {
    Error handling
}

if (f_out.openf("w", "%s/%s", "directory", "file.txt.gz") < 0) {
    Error handling
}

while ((line_ptr = net_in.getline()) != NULL) {
    if (f_out.printf("%s", line_ptr) < 0) {
        Error handling
    }
}

f_out.close();
net_in.close();
```

8.8.2 openp(), openpf(), vopenpf()

NAME

openp(), openpf(), vopenpf() — Opens streams (Perl-like)

SYNOPSIS

```
int openp( const char *path );
int openpf( const char *path_fmt, ... );
int vopenpf( const char *path_fmt, va_list ap );
```

DESCRIPTION

These member functions are similar to `open()` in the scripting language Perl, with `path` or `path_fmt` indicating the file or URL to be opened. If it used to indicate a command it executes that command and then connects the result using a pipe to open the stream.

Whether read or write mode is used is expressed by adding "`<`" or "`>`", as in `"< infile.txt"` and `"> outfile.txt"`, when `path` or `path_fmt` is used indicate the file or URL. If "`<`" is used the stream is opened in read mode (Refer to EXAMPLE 1), and if "`>`" is used the stream is opened in write mode. If neither "`<`" or "`>`" is specified the stream is opened in read mode. If a compressed file is specified the file gets automatically compressed/expanded.

If `path` or `path_fmt` is used to indicate a command, "`|`" needs to be placed before or after `path` to express the command, as in `"command |"` or `"| command"`. "`|`" being placed after `path`results in the command specified being executed, and the output from standard output of the command being connected with a pipe so that it can be read by reading member functions of the object (Refer to EXAMPLE 2). "`|`" being placed before "`path` results in the specified command being executed, and the result written by writing member functions of the object is then connected with a pipe so that it can be read from the standard input for the command (Refer to EXAMPLE 3). `path` or `path_fmt` being the command results in execution in the "`/bin/sh -c` *command*" format, and hence `path` can include pipe or redirection symbols (`|`,`<`,`>`) (Refer to EXAMPLE 2).

When `path` or `path_fmt` is `NULL` the standard input is used.

For more details on the arguments for `path_fmt` and later for `openpf()` and `vopenpf()`, refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`path`	File, URL or command
[I]	`path_fmt`	File, URL or command format specifications
[I]	`...`	Each element of data of a file, URL or command
[I]	`ap`	All the elements of data of a file, URL or command

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (Error)	:	If the stream has already been opened by any of the other member functions described in this section.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the system failed to open a file, URL or command.

EXCEPTION

If the system failed to generate an object.

EXAMPLE-1

Connects to the `path` using port number `port` for server name `server` in read only mode with http protocol, and then writes the content of that stream to standard output:

```
digeststreamio net_in;
digeststreamio s_io;
const char *line_ptr;

if (net_in.openpf("< http://%s:%d%s",server,port,path) < 0) {
    Error handling
}

if (s_io.open("w") < 0) {
    Error handling
}

while ((line_ptr = net_in.getline()) != NULL) {
    if (s_io.printf("%s", line_ptr) < 0) {
        Error handling
    }
}

s_io.close();
net_in.close();
```

EXAMPLE-2

The following code sorts the content of *file.txt* in *directory* using the second field as the key and on a per-line basis, and then prints the result to standard output through the open process.

The argument for `openp()` is specified in the `"command |"` format, where the first "`|`"indicates a pipe and the second "`|`" that the argument is a command.

This code does the same thing as in EXAMPLE-2 in §8.7.1 but with "`|`" placed after `command`.

```
digeststreamio p_in;
const char *line_ptr;

if (p_in.openp("cat directory/file.txt | sort -k 2 |") < 0) {
    Error handling
}

while ((line_ptr = p_in.getline()) != NULL) {
    printf("%s",line_ptr);
}

p_in.close();
```

EXAMPLE-3

The following code reads the content of *file.txt* in *directory* line by line, and then displays it on the screen using the less command:

```
stdstreamio f_in;
digeststreamio p_out;
const char *line_ptr;

if (f_in.open("r", "directory/file.txt") < 0) {
    Error handling
}

if (p_out.openp("| less") < 0) {
    Error handling
}

while ((line_ptr = f_in.getline()) != NULL) {
    p_out.printf("%s",line_ptr);
}

p_out.close();
f_in.close();
```

8.8.3 is_write_mode()

NAME

is_write_mode() — Returns mode of opened stream

SYNOPSIS

```
bool is_write_mode() const;
```

DESCRIPTION

This member function returns `true` for opened stream in write mode, otherwise returns `false`.

8.8.4 content_length()

NAME

content_length() — Calls for the length of streams

SYNOPSIS

```
long long content_length() const;
```

DESCRIPTION

If the stream opened by `open()` has information about length of stream, `content_length()` returns the byte length of the stream. With compressed streams, it returns the byte length of a compressed stream.

RETURN VALUE

Non-negative value	:	Byte length of stream
Negative value (error)	:	If information of length does not exist.

EXAMPLE

The following code opens the URL of *http://www.jaxa.jp/* in read mode, and then prints the byte length of the stream to standard output.

```
digeststreamio net_in;
long long l_ret;

if (net_in.open("r", "http://www.jaxa.jp/") < 0) {
    Error handling
}

if ((l_ret = net_in.content_length()) < 0) {
    printf("No information about stream byte size\n");
}
else {
    printf("Stream Byte Size = %lld \n", l_ret);
}

net_in.close();
```

8.8.5 user_agent().assign()

NAME

user_agent().assign() — Sets user agent

SYNOPSIS

```
tstring &user_agent().assign( const char *uagent );
tstring &user_agent().assignf( const char *uagent_fmt, ... );
```

DESCRIPTION

Sets the user agent to be transmitted when connecting to a Web server. If a user agent is not set, "hostname SLLIB-x.x::httpstreamio" will be transmitted.

8.8.6 username().assign()

NAME

username().assign() — Sets the name for FTP user

SYNOPSIS

```
tstring &username().assign( const char *user );
```

DESCRIPTION

By default, the `open()` member function enables users to log on to an FTP server anonymously, but a user name can be set using this member function.

8.8.7 password().assign()

NAME

password().assign() — Sets the password for FTP user

SYNOPSIS

```
tstring &password().assign( const char *pass );
```

DESCRIPTION

Sets the password for an FTP user assigned by `username().assign()`.

8.9 The TERMLINEIO class

The termlineio class is used to help users input commands from the GNU readline library. It supports a cursor key and history function etc when commands are input. It inherits cstreamio and hence all the member functions in §8.1 are available for use (You do not need to learn the GNU readline APIs). With the termlineio class `open()` and `close()` must always be used. With the `open()` member function `"r"` or `"w"` is specified as the `mode`. The `mode` being `"r"` results in commands being input on a per-line basis, and if `"w"` commands are output to the pager. The pager specified by the environment variable `PAGER` is utilized.

If you wish to use the termlineio class you must add "`#include <sli/termlineio.h>`" to the code. In addition, if you need to declare a namespace (§4.1)) you must also add "`using namespace sli;`" to the code.

Table 15 lists the member functions. The following table provides the member functions with the same corresponding features as in libc.

	The termlineio class	Feature	Corresponding function in libc
§8.9.1	`open()`, `openf()`, `vopenf()`	Opens streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of single character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of single line	—
§8.1.11	`scanf()`	Converts inputs with a format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of single character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of a buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.9.2	`set_prompt()`	Sets prompt	—
§8.9.3	`automate_history()`	Specifies whether to automatically save histories	—
§8.9.4	`add_history()`	Adds command histories to history buffer	—
§8.9.5	`clear_history()`	Initialization of history buffers	—
§8.9.6	`stifle_history()`	Restricts number of history buffers	—
§8.9.7	`unstifle_history()`	Removes restriction on number of history buffers	—
§8.9.8	`read_history()`	Reads histories from a file	—
§8.9.9	`write_history()`	Writes histories to a file	—

Table 15: List of the member functions available for use with the termlineio class.

How to use the member functions redefined and member added in the termlineio class is described below.

8.9.1 open()

NAME

open(), openf(), vopenf() — Opens streams

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, const char *path );  ..... 3
int open( const char *mode, const char *const argv[] );  ..... 4
int openf( const char *mode, const char *path_fmt, ... );  ..... 5
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 6
```

DESCRIPTION

The `mode` being `"r"` or `"r+"` results in the input from the terminal using the GNU readline. `path` can be used to specify the file in which the history is saved. The `"r"` specification can be used to specify that the content of the history buffer is not saved to the file upon `close()`, while with the `"r+"` specification the history is saved to the file upon `close()`.

The `mode` being `"w"` performs output to the pager. The pager is specified by `path` or `argv`, but if not specified the pager specified by the environment variable `PAGER` is utilized.

`path` or `path_fmt` being set results in execution in the "`/bin/sh -c` *command*" format, and hence `path` or `path_fmt` can include the pipe or redirection symbols (`|`,`<`,`>`).

The elements of `argv[]` must be set in the order of "*executable file's path name, argument 1, argument 2, ...* `NULL`". The end of `argv[]` must always be `NULL`.

Member functions 1 and 2 cannot execute commands and connect the result using a pipe to open the stream. For more details on these member functions refer to the descriptions provided in §8.1.1.

For more details on the arguments for `path_fmt` and later for member functions 5 and 6 refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	File or pager opening mode
[I]	`fd`	File descriptor
[I]	`path`	File or pager
[I]	`argv[]`	All the elements of data of a file or pager
[I]	`path_fmt`	File or pager format specifications
[I]	`...`	All the elements of data of a file or pager
[I]	`ap`	All the elements of data of a file or pager

([I] : input, [O] : output)

RETURN VALUE

0	:	Normal termination
Negative value (Error)	:	If open mode was not set.
	:	If the system failed to open a stream
	:	If the system failed to open a stream because it cannot access the stream in the specified mode etc.
	:	If the system failed to open a stream because the relationship between the `mode` specified and `fd` was incorrect etc (Member function 2).
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the other member functions described in this section.

EXCEPTION

If the system failed to secure the buffer for reading data (Member functions 1, 3, 5, and 6).
If the system failed to generate a process (Other than member function 2).
If the system failed to generate a file descriptor for reading and writing data (Other than member function 2).
If any of the elements of data of a file or pager does not meet the format specifications (Member functions 5, and 6).

EXAMPLE

The following code opens *command_history.txt* in read and write mode, and reads it to the history buffer. It then reads to the history buffer the line that is input from the command line, and writes it to *command_history.txt*:

```
termlineio t_in;

if ( t_in.open("r+", "command_history.txt") < 0 ) {
    Error handling
}

t_in.getline();

t_in.close();
```

8.9.2 set_prompt(), setf_prompt(), vsetf_prompt()

NAME

set_prompt(), setf_prompt(), vsetf_prompt() — Sets a prompt

SYNOPSIS

```
termlineio &set_prompt( const char *prompt );  ............................. 1
termlineio &setf_prompt( const char *prompt_fmt, ... );  .................. 2
termlineio &vsetf_prompt( const char *prompt_fmt, va_list ap );  ......... 3
```

DESCRIPTION

Sets the prompt displayed when inputting commands.

Member function 1 sets `prompt`.

Member functions 2 and 3 set to the prompt a string that is converted depending on the conversion specifications provided in `format`. For more details on the arguments for `format` and later refer to the descriptions provided in §8.1.14.

PARAMETER

[I]	`prompt`	Prompt
[I]	`prompt_fmt`	Prompt format specifications
[I]	`...`	Each element of data for a prompt
[I]	`ap`	All the elements of data for a prompt

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure the buffer for setting a prompt.
If each element of data of a prompt does not meet the format specified for the prompt (Member functions 2 and 3).

EXAMPLE

Sets the string that displays the prompt. When this code is executed, you will be prompted by `"prompt> "` for one-line command input:

```
termlineio t_in;
const char *cmd;

if ( t_in.open("r") < 0 ) {
    Error handling
}

cmd = t_in.set_prompt("prompt> ").getline();
printf("Your command is '%s'\n", cmd);

t_in.close();
```

8.9.3 automate_history()

NAME

automate_history() — Specifies whether to automatically save histories

SYNOPSIS

```
termlineio &automate_history( bool tf );
```

DESCRIPTION

If the Auto-save History flag `tf` is `true`, all non-empty input lines are added to the history buffer. If `tf` is `false`, no input lines are registered to the history buffer. To register input lines to the history buffer after instructing not to register input lines to the history buffer using this member funciton, you will need to use the `add_history()` member function (§8.9.4). Please note that the default value for `tf` is `true`.

PARAMETER

[I] `tf` Auto-save History flag (true/false)
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXAMPLE

The following code sets the Auto-save History flag to false and then reads the command line three times. The up-arrow key can be pressed while it is being read to verify that the input command has not been registered, and that only the strings included in the read file of *command_history.txt* have been registered:

```
termlineio t_in;
int        i_count = 0;
```

```
if ( t_in.open("r", "command_history.txt") < 0 )  {
    Error handling
}

/* Does not auto-save histories */
t_in.automate_history( false );

for (i_count = 0 ; i_count < 3 ; i_count++){
    t_in.getline();
}

t_in.close();
```

8.9.4 add_history()

NAME

add_history() — Adds commands to history buffer

SYNOPSIS

```
termlineio &add_history( const char *line );
```

DESCRIPTION

Adds a command `line` to history buffer. This member function can be used after instructing not to automatically save histories using the `automate_history()` member function (§8.9.3.

PARAMETER

[I] `line` Name of command
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure the buffer for copying a command.
If the system failed to store a command to the history buffer.

EXAMPLE

This code sets the Auto-save History flag to false, and then executes the input command with system() function. The command must have normally terminated for the command to be added to the history buffer using the `add_history()` member function.
The content of the history buffer is saved in *command_history.txt*.

```
termlineio t_in;
const char *cmd;

if ( t_in.open("r+", "command_history.txt") < 0 )  {
    Error handling
}

/* Does not auto-save histories */
t_in.automate_history( false );

/* Accept the command until Ctrl-D is pressed */
while ( (cmd=t_in.getline()) != NULL ) {
    /* Execute the command using system(), and saves the history only when the command has terminated n
    if ( system(cmd) == 0 ) {
```

```
            t_in.add_history(cmd);
        }
    }

    t_in.close();
```

8.9.5 clear_history()

NAME

clear_history() — Initialization of history buffers

SYNOPSIS

```
termlineio &clear_history();
```

DESCRIPTION

Deletes all the content of a history buffer.

RETURN VALUE

Reference to itself

EXAMPLE

This code clears all the histories that have been registered. It first reads the input from the command line three times and then registers it to the history buffer. Upon `"check history> "` being output the up-arrow key can be pressed to verify that all the history buffers have been cleared.

```
termlineio t_in;
int        i_count = 0;

if ( t_in.open("r") < 0 )  {
    Error handling
}

for (i_count = 0 ; i_count < 3 ; i_count++){
   t_in.getline();
}

t_in.clear_history();
t_in.set_prompt("check history >").getline();

t_in.close();
```

8.9.6 stifle_history()

NAME

stifle_history() — Restricts the number of history buffers

SYNOPSIS

```
termlineio &stifle_history( int num_lines );
```

DESCRIPTION

Restricts the number of history buffers to the maximum of `num_lines`. This restriction can be removed using the `unstifle_history()` member function (§8.9.7).

PARAMETER

[I] `num_lines` Number of history buffers

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXAMPLE

This code restricts the history buffers that can be registered, with the maximum number of history buffers that can be registered being 3. The command line is then read five times, and the command registered to the history buffer. Upon `"check history> "` being output the up-arrow key can be pressed to verify that the first two histories registered have been cleared.

```
termlineio t_in;
int        i_count = 0;

if ( t_in.open("r") < 0 )  {
    Error handling
}

/* Restrict the history buffers to 3 */
t_in.stifle_history(3);

for (i_count = 0 ; i_count < 5 ; i_count++){
   t_in.getline();
}

t_in.set_prompt("check history >").getline();

t_in.close();
```

8.9.7 unstifle_history()

NAME

unstifle_history() — Removes restriction on number of history buffers

SYNOPSIS

```
termlineio &unstifle_history();
```

DESCRIPTION

Removes the restriction on the number of history buffers that can be set using `stifle_history()` (§8.9.6).

RETURN VALUE

Reference to itself

EXAMPLE

This code removes the history buffer restriction. It first performs the same processing as in the `EXAMPLE` provided in §8.9.6, then removes the history buffer restriction, and registers the command to the history buffer again. Upon `"check history2> "` being output the up-arrow key can be pressed to verify that eight command histories have been registered.

```
termlineio t_in;
int        i_count = 0;

if ( t_in.open("r") < 0 )  {
    Error handling
}

/* Restrict the history buffers to 3 */
t_in.stifle_history(3);

for ( i_count = 0 ; i_count < 5 ; i_count++){
   t_in.getline();
}

t_in.set_prompt("check history1 >").getline();
t_in.set_prompt("");

/* Remove restriction on history buffers  */
t_in.unstifle_history();

for ( i_count = 0 ; i_count < 5 ; i_count++){
   t_in.getline();
}

t_in.set_prompt("check history2 >").getline();

t_in.close();
```

8.9.8 read_history(), readf_history(), vreadf_history()

NAME

read_history(), readf_history(), vreadf_history() — Reading of histories from a file

SYNOPSIS

```
read_history( const char *path );  ..... 1
readf_history( const char *path_fmt, ... );  ..... 2
vreadf_history( const char *path_fmt, va_list ap );  ..... 3
```

DESCRIPTION

Reads the file specified by `path` and then adds the content of the file to the history buffer. Member functions 2 and 3 are used to convert the file depending on the conversion specifications in `format`, and creates a name for the file to read. For more details on the arguments for `format` and later refer to the descriptions provided in §8.1.14.

PARAMETER

[I]	`path`	Name of file
[I]	`path_fmt`	File name format specifications
[I]	`...`	Each element of data of a file name
[I]	`ap`	All the elements of data of a file name

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Other than 0 (Error)	:	If the system failed to add a command to the history buffer because the file etc specified does not exist etc.

EXCEPTION

If each element of data of a file path does not meet the format specifications (Member functions 2 and 3).

EXAMPLE

This code adds commands that are read from *temporary_file.txt* to the history buffer. Upon the `"push ^ button >"` prompt being output the up-arrow key can be pressed to verify that the content of *temporary_file.txt* has been saved in the history buffer. As a precondition, strings in *command_history.txt* and *temporary_file.txt* must be included in advance.

```
termlineio t_in;

if (t_in.open("r", "command_history.txt") < 0) {
    Error handling
}

if (t_in.read_history("temporary_file.txt") != 0) {
    Error handling
}
else {
    t_in.set_prompt("push ^ button >").getline();
}

t_in.close();
```

8.9.9 write_history(), writef_history(), vwritef_history()

NAME

write_history(), writef_history(), vwritef_history() — Writing of histories to a file

SYNOPSIS

```
int write_history( const char *path );  ..... 1
int writef_history( const char *path_fmt, ... );  ..... 2
int vwritef_history( const char *path_fmt, va_list ap );  ..... 3
```

DESCRIPTION

Writes the content of a history buffer to the file specified by `path`. Creates a new file if the file does not exist. If the file specified already exists, it is overwritten. If NULL is specified as the file name it saves histories to `~/.history`.

Member functions 2 and 3 create a name for the file to write to, converted depending on the conversion specifications privided in `format`. For more details of the arguments for `format` and later refer to the descriptions in §8.1.14.

PARAMETER

[I]	`path`	Name of file
[I]	`path_fmt`	File name format specifications
[I]	`...`	Each element of data of a file name
[I]	`ap`	All the elements of data of a file name

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Other than 0 (Error)	:	If the system failed to write a history to the file because you do not have the authority to execute the file etc specified etc.

EXCEPTION

If each element of data of a file path does not meet the format specifications (Member functions 2, 3)

EXAMPLE

This code writes the content of a history buffer to *new_file.txt*. The content of the file *command_history.txt* that is read to the history buffer is written to *new_file.txt*. As a precondition, strings in the file *command_history.txt* must be included in advance.

```
termlineio t_in;

if (t_in.open("r", "command_history.txt") < 0) {
    Error handling
}

if (t_in.write_history("new_file.txt") != 0) {
    Error handling
}

t_in.close();
```

8.10 The TERMSCREENIO class

The termscreenio class is used to perform input (using temporary files) through an editor and output to a pager on a terminal. The environment variables `EDITOR` and `PAGER` are referenced internally. It inherits cstreamio and hence all the member functions in §8.1 are available for use with the class. With the termscreenio class, `open()` and `close()` must always be used.

If you wish to use the termscreenio class, you must add "`#include <sli/termscreenio.h>`" to the code. In addition, if you need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

Table 16 lists the member functions. Member functions that have the same functions as in the libc are provided in the table.

	The termscreenio class	Feature	Corresponding function in libc
§8.10.1	`open()`, `openf()`, `vopenf()`	Opens streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of single character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of single line	—
§8.1.11	`scanf()`	Converts inputs with a format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of single character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of a buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.

Table 16: List of the member functions available for use with the termscreenio class.

How to use the member functions redefined and added in the termscreenio class is described below.

8.10.1 open()

NAME

open() — Opens streams

SYNOPSIS

```
int open( const char *mode );  ..... 1
int open( const char *mode, int fd );  ..... 2
int open( const char *mode, const char *path );  ..... 3
int open( const char *mode, const char *const argv[] );  ..... 4
int openf( const char *mode, const char *path_fmt, ... );  ..... 5
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 6
```

DESCRIPTION

The `mode` being `"r"` results in a temporary file being created and the editor activated. Closing the editor then results in the edited temporary file being opened as a stream. The editor can be specified by `path` or `argv`, and if not specified is activated as indicated by the environment variable `EDITOR`. If the environment variable is not set, `vi` is activated.

The `mode` being `"w"` results in the stream being opened by the pager. The pager can be specified by `path` or `argv`, and if not specified is activated as indicted by the environment variable `PAGER`. If the environment variable is not set, `more` is activated.

The elements of `argv[]` must be set in the order of *executable file's path name argument 1, argument 2, ...* `NULL`. The end of `argv[]` must always be `NULL`.

Member functions 1 and 2 cannot open streams through an editor or pager. For more details on these member functions refer to the descriptions provided in §8.1.1.

For more details on the arguments for `path_fmt` and later for member functions 5 and 6, refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	Editor or pager opening mode
[I]	`fd`	File descriptor
[I]	`path`	Editor or pager
[I]	`argv[]`	All the elements of data of an editor or pager
[I]	`path_fmt`	Editor or pager format specifications
[I]	`...`	Each element of data of an editor or pager
[I]	`ap`	All the elements of data of an editor or pager

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (Error)	:	If open mode was not set.
	:	If the system failed to open a stream.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the stream has already been opened by any of the other member functions described in this section.

EXCEPTION

If the system failed to secure the buffer for reading data (Member functions 1, 3, 5, and 6).
If the system failed to generate a process (Other than member function 2).
If the system failed to generate a file descriptor for reading and writing data (Other than member function 2).
If each element of data of an editor or pager does not meet the format specifications (Member functions 5 and 6).

EXAMPLE

The following code performs input from the editor vi activated in binary mode, and then displays the first line input as standard output:

```
termscreenio t_in;

if (t_in.open("r", "vi -b") < 0) {
    Error handling
}

printf("%s\n", t_in.getline());
```

```
t_in.close();
```

8.11 The INETSTREAMIO class

It inherits cstreamio and hence all the member functions in §8.1 are available for use with the class. This class is used to implement the httpstreamio (§8.5) and ftpstreamio (§8.6) classes.

With the inetstreamio class, `open()` and `close()` must always be used. With the open() member function, `"r+"` is specified as the `mode` and a URL, for example `"http://www.jaxa.jp/"`, specified as the `path`. Resolves the port number and host name to use with the URL, and then connects to the server. Connections through a proxy server are not supported.

If you wish to use the inetstreamio class you must add "`#include <sli/inetstreamio.h>`" to the code. In addition, if you need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

Table 17 lists the member functions. Member functions that have the same feature as in libc are provided in the table.

	The inetstreamio class	Feature	Corresponding function in libc
§8.11.1	`open()`, `openf()`, `vopenf()`	Opens streams	`fopen()`
§8.1.2	`close()`	Closes streams	`fclose()`
§8.1.3	`read()`	Input of binary streams	`fread()`
§8.1.3	`write()`	Output of binary streams	`fwrite()`
§8.1.4	`bread()`	Input of binary streams (With endian conversion)	—
§8.1.5	`bwrite()`	Output of binary streams (With endian conversion)	—
§8.1.6	`rskip()`	Read n bytes to skip data stream	—
§8.1.7	`wskip()`	Write n bytes of blank data	—
§8.1.8	`getchr()`	Input of single character	`fgetc()`
§8.1.9	`getstr()`	Input of strings	`fgets()`
§8.1.10	`getline()`	Input of single line	—
§8.1.11	`scanf()`	Converts inputs with a format and assigns to an argument	`fscanf()`
§8.1.12	`putchr()`	Output of single character	`fputc()`
§8.1.13	`putstr()`	Output of strings	`fputs()`
§8.1.14	`printf()`	Outputs the value of an argument after being format-converted	`fprintf()`
§8.1.15	`flush()`	Forcefully outputs the content of a buffer	`fflush()`
§8.1.16	`eof()`, `error()`, `reseterr()`	Check and reset stream status	`feof()`, etc.
§8.11.2	`path()`	Returns the path specified by a URL	—
§8.11.3	`host()`	Returns the path specified by a URL	—

Table 17: List of the member functions available for use with the inetstreamio class.

How to use the member functions redefined and added in the inetstreamio class is described below.

8.11.1 open()

NAME

open() — Opens streams

SYNOPSIS

```
int open( const char *mode, const char *path );  ..... 1
int openf( const char *mode, const char *path_fmt, ... );  ..... 2
int vopenf( const char *mode, const char *path_fmt, va_list ap );  ..... 3
```

DESCRIPTION

Opens the URL indicated by `path` or `path_fmt`. If `path` or `path_fmt` is `NULL`, the standard input or standard output is used.

`"r"`, `"r+"`, `"w"` or `"w+"` can be specified as the `mode`. `"r"` and `"w"` result in a read-only and write-only one-way connection, respectively. `"r+"` and `"w+"` result in two-way connections that allow both reading and writing. `"r+"` and `"w+"` result in the same behavior as each other.

For more details on the arguments for `path_fmt` for the member functions 2 and 3 refer to the descriptions provided in §8.1.1.

PARAMETER

[I]	`mode`	URL opening mode
[I]	`path`	Path of URL
[I]	`path_fmt`	URL format specifications
[I]	`...`	Each element of data of a URL
[I]	`ap`	All the elements of data of a URL

([I] : Input, [O] : Output)

RETURN VALUE

0	:	Normal termination
Negative value (Error)	:	If the system failed to open a stream because a URL was not specified etc.
	:	If the system failed to open a stream because the specified URL was inappropriate etc.
	:	If the string indicating the path for `path_fmt` exceeds `PATH_MAX`.
	:	If the system failed to open a stream because the `mode` was not specified etc.
	:	If the system failed to open a stream because the specified `mode`was appropriate etc.
	:	If the system failed to open a stream because it cannot access the stream in the specified mode etc.
	:	If the stream has already been opened by any of the other member functions described in this section.

EXCEPTION

If the system failed to allocate enough memory.
If the system failed to establish a socket connection.
If the system failed to open a socket in the specified mode.
If all the elements of data of a URL do not meet the format specifications (Member functions 2 and 3).

EXAMPLE

For an EXAMPLE refer to §8.11.4.

8.11.2 path()

NAME

path() — Returns the path specified by a URL

SYNOPSIS

```
const char *path();
```

DESCRIPTION

Returns a string for the path part extracted from `path` as specified by the open() member function (§8.11.1).

RETURN VALUE

Path for URL

EXAMPLE

For an EXAMPLE refer to §8.11.4.

8.11.3 host()

NAME

host() — Returns the host name specified by a URL

SYNOPSIS

```
const char *host();
```

DESCRIPTION

Returns a string for the host name extracted from `path` as specified by the open() member function (§8.11.1).

RETURN VALUE

Host name specified by URL

EXAMPLE

For an EXAMPLE refer to §8.11.4.

8.11.4 Sample code

Here is an example of a simple HTTP client.

```
#include <sli/stdstreamio.h>
#include <sli/inetstreamio.h>
using namespace sli;

int main()
{
    int status = -1;
    stdstreamio sio;
    inetstreamio isio;
    const char *line_ptr;
    /* Connects to http server */
    if ( isio.open("r+","http://www.jaxa.jp/") < 0 ) {
        sio.eprintf("[ERROR] isio.open() failed\n");
        goto quit;
    }
    /* Send request to http server */
    isio.printf("GET %s HTTP/1.0\r\n",isio.path());
    isio.printf("User-Agent: My Program\r\n");
    isio.printf("Host: %s\r\n",isio.host());
    isio.printf("Connection: close\r\n");
    isio.printf("\r\n");
    isio.flush();
    /* Receive data from http server  */
    while ( (line_ptr=isio.getline()) != NULL ) {
        sio.printf("%s",line_ptr);
    }
    /* Terminate connection */
    isio.close();
    status = 0;
quit:
    return status;
}
```

9 The TSTRING class

The tstring class provides APIs that enable users to execute the string processing seen in scripting languages such as Perl, PHP and Ruby, along with the string processing that is extremely similar to the functions provided by `stdio.h`, `string.h`, `strings.h`, `stdlib.h` and `ctype.h` in libc.

A wealth of APIs are available for use, ranging from a member function that sets characters to a given position character by character to a member function that allows users to use POSIX extended regular expressions. tstring class member functions use a very regular order in their arguments, and hence can be learned quite easily.

String buffers and their size are automatically managed internally and hence when editing a string users do not need to create a buffer or worry about the size of a buffer. For example, you can assign a string immediately after creating an object, as shown below:

```
    tstring my_str;                     /* Create object */
    my_str.printf("Hello World");       /* Assign "Hello World" to my_str */
    sio.printf("%s\n",my_str.cstr());   /* Output content of my_str to STDOUT */
```

If you wish to use the tstring class you must add "`#include <sli/tstring.h>`" to the code. In addition, if you need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

The following provides an example that is close to an actual case of it being used.

```
#include <sli/stdstreamio.h>
#include <sli/tstring.h>
using namespace sli;

int main()
{
    stdstreamio sio;
    stdstreamio fin;
    const char *line;

    fin.open("r","infile.txt");
    while ( (line=fin.getline()) != NULL ) {
        tstring str0;
        /* Store a line, and remove spaces, tabs and newline chars from both ends */
        str0.assign(line).trim(" \t\n");
        /* When there is one or more character, ... */
        if ( 0 < str0.length() ) {
            if ( str0.cchr(0) == '#' ) {
                /* Display comment */
                sio.printf("%s\n",str0.cstr());
            }
            else {
                /* Convert to integer value */
                int n, a=0, b=0;
                n = str0.scanf("%d %d",&a,&b);
                sio.printf("n=%d  a=%d b=%d\n",n,a,b);
            }
        }
    }
    fin.close();

    return 0;
}
```

In this example `infile.txt` is opened, the line beginning with # displayed as a comment, and if

there is anything else but a space, tab or newline character it is input in the format to retrieve the value, and then displayed.

Please note that in this section the `cstreamio` class (§8)) is used in the EXAMPLES, as in the example above.

9.1 Creating an object —three operating modes

There are three operating modes for use with tstring class objects. The operating mode must be carefully selected depending on the purpose. Please note that the operating mode can only be determined when an object is first created.

9.1.1 Normal mode

If nothing is specified when an object is created, as in the following example:

```
tstring my_str;
```

The string inside does not have a buffer for the object created, and the return value for the cstr() member function (§9.5.3) is `NULL`. This will be referred to as the **normal mode**. However, when a member function (except `init()`) is used to modify a strings the object needs to always retain a string (`""`, at minimum length). If you do wish to to make an object with the string "empty" or `NULL`, the init() member function (§9.5.12) can be used or `NULL` assigned using the operator "`=`" (§9.4.2).

In normal mode an initial value can also be provided, as in the following:

```
tstring my_str("Hello");
```

9.1.2 NULL-free mode

Problems can occur when the string inside is `NULL`, but to avoid that you can set the flag to be `true` when creating an object, as in the following:

```
tstring my_str(true);
```

The object always retains a string with this, and `NULL` will not be returned by the cstr() member function (§9.5.3). This will be referred to as the **NULL-free mode**.

9.1.3 Fixed-length buffer mode

Another use is **fixed-length buffer mode**, which provides a method of specifying the maximum length of strings of an argument that you wish to handle and when creating an object, as in the following:

```
tstring my_str(64);
```

In the above example the object can handle strings with a maximum of 64 characters. In this mode only strings equal to or shorter than the string length initially specified can be handled, with the member functions for editing strings being designed to maintain the memory to be secured again at a minimum and allow code to run at high speed. With the fixed-length buffer mode `NULL` will not be returned by the cstr() member function (§9.5.3).

9.1.4 Restriction with fixed-length buffer mode

With the normal and NULL-free modes, member functions that modify strings inside an object can have arguments provided with a reference to the object itself or the address returned by `cstr()`,

whereas the fixed-length buffer mode does not allow for this type of use as it assigns the highest priority to operating speed.

9.2 Regularity of arguments for member functions

When the position and number of characters for a string inside an object are provided in the argument specifications they should always appear at the front of an argument. For example, with the `strtol()` member function (§9.5.37) `pos` and `n` appear on the left and show the position and length of the string inside the object, as in the following:

```
long strtol( int base, size_t *endpos ) const;
long strtol( size_t pos, int base, size_t *endpos ) const;
long strtol( size_t pos, size_t n, int base, size_t *endpos ) const;
```

It could also be said that the arguments on the left provide the specifications for objects, while the arguments on the right provide specifications for anything but an object.

9.3 List of member functions

Table 18 lists the member functions. Member functions that have the same feature as in libc are provided in the table.

	Name of member function	Feature	Corresponding function in libc
§9.4.1	`[]`	Reference to characters in a specified position	—
§9.4.2	`=`	Assigns strings	—
§9.4.3	`+=`	Addition of strings	—
§9.4.4	`==`	Comparison of strings	—
§9.4.5	`!=`	Comparison of strings	—
§9.5.1	`length()`	Length of a string	`strlen()`
§9.5.2	`max_length()`	Maximum length of a string	—
§9.5.3	`cstr()`, `c_str()`	Beginning address for a string (read-only)	—
§9.5.4	`str_ptr()`, `str_ptr_cs()`	Beginning address for a string	—
§9.5.5	`cchr()`	Reading of characters in a specified position	—
§9.5.6	`at()`, `at_cs()`	Reference to characters in a specified position	—
§9.5.7	`update_length()`	Update internal information (Use this when directly writing data to internal buffer of fixed-length buffer mode.)	—
§9.5.8	`dprint()`	Outputs object information to standard error output	—
§9.5.9	`getstr()`	Copies (sub)string to external buffer	—
§9.5.10	`copy()`	Copies (sub)string to external object	`strdup()`
§9.5.11	`swap()`	Swaps objects	—
§9.5.12	`init()`	Complete initialization of objects	—
§9.5.13	`printf()`, `assignf()`	Initialization of objects	`sprintf()`
§9.5.14	`implode()`	Sets a string that elements of a string array joined with delimiter	—
§9.5.15	`import_binary()`	Import of binary data	—
§9.5.16	`put()`, `putf()`	Sets characters or strings to a given position	—
§9.5.17	`strcat()`, `append()`	Addition of characters or strings	`strcat()`
§9.5.17	`strncat()`, `append()`	Addition of characters or strings	`strncat()`
§9.5.18	`insert()`, `insertf()`	Insertion of characters or strings	—
§9.5.19	`replace()`, `replacef()`	Replacement of strings	—
§9.5.20	`erase()`	Erasure of strings	—
§9.5.21	`clean()`	Pads existing whole strings with a given character	—
§9.5.22	`resize()`	Changes the length of strings	—
§9.5.23	`resizeby()`	Changes the relative length of strings	—
§9.5.24	`crop()`	Crops strings	—
§9.5.25	`chomp()`	Elimination of newline characters	—
§9.5.26	`trim()`	Elimination of spaces on both ends of a string	—
§9.5.27	`ltrim()`	Elimination of a space to the left of a string	—
§9.5.28	`rtrim()`	Elimination of a space to the right of a string	—
§9.5.29	`strreplace()`	Searches and replaces strings	—
§9.5.30	`regreplace()`	Replaces parts that match an extended regular expression	—
§9.5.31	`tolower()`	Converts uppercase to lowercase characters	`tolower()`
§9.5.32	`toupper()`	Converts lowercase to uppercase characters	`toupper()`
§9.5.33	`expand_tabs()`	Replaces TAB characters with white space characters	—
§9.5.34	`contract_spaces()`	Replaces white space characters with TAB characters	—
§9.5.35	`atoi()`	Converts to integer value	`atoi()`
§9.5.35	`atol()`	Converts to integer value	`atol()`
§9.5.35	`atoll()`	Converts to integer value	`atoll()`
§9.5.36	`atof()`	Converts to real value	`atof()`

Table 18: List of the member functions available for use with the tstring class (Continued on next page).

	Name of member function	Feature	Corresponding function in libc
§9.5.37	`strtol()`	Converts to integer value	`strtol()`
§9.5.37	`strtoll()`	Converts to integer value	`strtoll()`
§9.5.38	`strtoul()`	Converts to unsigned integer value	`strtoul()`
§9.5.38	`strtoull()`	Converts to unsigned integer value	`strtoull()`
§9.5.39	`strtod()`	Converts to real value	`strtod()`
§9.5.40	`scanf()`	Formatted input conversion	`sscanf()`
§9.5.41	`strcmp()`, `compare()`	Comparison of strings	`strcmp()`
§9.5.42	`strncmp()`, `compare()`	Partially compares strings	`strncmp()`
§9.5.43	`strcasecmp()`	Comparison of strings (Case-independent)	`strcasecmp()`
§9.5.43	`strncasecmp()`	Comparison of strings (Case-independent)	`strncasecmp()`
§9.5.44	`isalnum()`	Inquires whether alphabetical or numerical character	`isalnum()`
§9.5.44	`isalpha()`	Inquires whether alphabetical character	`isalpha()`
§9.5.44	`iscntrl()`	Inquires whether control character	`iscntrl()`
§9.5.44	`isdigit()`	Inquires whether numerical character (0 to 9)	`isdigit()`
§9.5.44	`isgraph()`	Inquires whether displayable character	`isgraph()`
§9.5.44	`islower()`	Inquires whether lowercase character	`islower()`
§9.5.44	`isprint()`	Inquires whether displayable character (Spaces included)	`isprint()`
§9.5.44	`ispunct()`	Inquires whether displayable character (Spaces and alphanumerics excluded)	`ispunct()`
§9.5.44	`isspace()`	Inquires whether white space character	`isspace()`
§9.5.44	`isupper()`	Inquires whether uppercase character	`isupper()`
§9.5.44	`isxdigit()`	Inquires whether hexadecimal number	`isxdigit()`
§9.5.45	`strchr()`, `find()`	Searches for character from left	`strchr()`
§9.5.46	`strstr()`, `find()`	Searches for a string from the left side	`strstr()`
§9.5.47	`strrchr()`, `rfind()`	Searches for a character from the right side	`strrchr()`
§9.5.48	`strrstr()`, `rfind()`	Searches for a string from right	—
§9.5.49	`find_first_of()`	Detects from the left any characters contained in a character set	`strpbrk()`
§9.5.50	`find_last_of()`	Detects from the right any characters contained in a character set	—
§9.5.51	`find_first_not_of()`	Detects from the left any characters not contained in a character	—
§9.5.52	`find_last_not_of()`	Detects from the right any characters not contained in a character	—
§9.5.53	`strpbrk()`	Detects from the left any characters contained in a character set	`strpbrk()`
§9.5.54	`strrpbrk()`	Detects from the right any characters contained in a character set	—
§9.5.55	`strspn()`	Inquires the length of characters contained in a character set run from the left	`strspn()`
§9.5.56	`strrspn()`	Inquires the length of characters contained in a character set run from the right	—
§9.5.57	`strcspn()`	Inquires the length of characters not contained in a character set run from the left	`strcspn()`
§9.5.58	`strmatch()`	Attempts Shell-like string matching	`fnmatch()`
§9.5.59	`regmatch()`	Attempts string matching with extended regular expression	`regexec()`

Table 18: List of member functions available for use with the tstring class (Continued from previous page).

9.4 Operators

Overuse of operators may reduce the readability of codes, and hence only the minimum have been made available in the library.

9.4.1 []

NAME

`[]` — Reference to the character in specified position

SYNOPSIS

```
unsigned char &operator[]( size_t pos );  ..... 1
const unsigned char &operator[]( size_t pos ) const;  ..... 2
```

DESCRIPTION

Returns reference to characters in position specified by `[]`.

Member function 1 can be used for both reading and writing and has the same behavior as `at()`, while member function 2 can only be used for reading and has the same behavior as `at_cs()`.

`pos` having a value longer than the string length specified to it results in the length of the string being automatically extended with member function 1, but with member function 2 an exception occurs.

Whether member function 1 or member function 2 is used is automatically determined by the presence or absence of the "const" attribute for an object. Member function 1 is automatically selected when the object does not have a "const" attribute and member function 2 when it does.

For more details on `at()` and `at_cs()` refer to the descriptions provided in §9.5.6.

PARAMETER

[I] `pos` Position of string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to the character in specified position

EXCEPTION

If `pos` has a value longer than the maximum string length specified to it in fixed-length buffer mode (Member function 1).
If `pos` has a value longer than the string length specified to it (Member function 2).

EXAMPLE

The following code reads the sixth character in a string that the object `my_str` includes, and then prints the result to standard output.
The character *X* is then written into the ninth position of the characters in `my_str`, and the result is printed to standard output:

```
stdstreamio   sio;
tstring       my_str = "abcdefgh";
unsigned char c_read;

c_read = my_str[6];
sio.printf("%c\n", c_read);
```

```
my_str[9] = 'X';
sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
g
abcdefgh␣X
```

(␣ refers to white space character.)

9.4.2 =

NAME

= — Assigns strings

SYNOPSIS

```
tstring &operator=(const tstring &obj);  ..... 1
const char *operator=(const char *str);  ..... 2
```

DESCRIPTION

Assigns the object or string specified to the right (argument) of the operator.

PARAMETER

[I] `obj` tstring class object
[I] `str` Address of string

RETURN VALUE

Reference to itself (Member function 1).
Address for internal buffer (Member function 2).

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory (Member function 1).

EXAMPLE

The following code assigns the string *Hello SLLIB User !* to a string that the object `my_str`includes, and prints the result to standard output. For more information on `c_str()` refer to the descriptions provided in §9.5.3.

```
stdstreamio sio;
tstring     my_str;

my_str = "Hello SLLIB User !";

sio.printf("%s\n", my_str.c_str());
```

Result of execution

```
Hello SLLIB User !
```

9.4.3 +=

NAME

+= — Addition of strings

SYNOPSIS

```
tstring &operator+=(const tstring &obj);  ..... 1
const char *operator+=(const char *str);  ..... 2
```

DESCRIPTION

Adds to a string the string specified to the right (argument) of the operator.

PARAMETER

[I] `obj` tstring class object
[I] `str` Address of string

RETURN VALUE

Reference to itself (Member function 1).
Address for internal buffer (Member function 2).

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code connects a string that the object `my_str` includes with the content of the string `c_sentence`, and then prints the result to standard output. For more information on `c_str()` refer to the descriptions provided in §9.5.3:

```
stdstreamio sio;
tstring     my_str      = "User ID : ";
const char  *c_sentence = "1234";

my_str += c_sentence;

sio.printf("%s\n", my_str.c_str());
```

Result of execution

```
User ID : 1234
```

9.4.4 ==

NAME

== — Comparison of strings

SYNOPSIS

```
bool operator==(const tstring &obj) const;
bool operator==(const char *str) const;
```

DESCRIPTION

Compares a string with the string specified to the right (argument) of the operator to verify whether they match.

PARAMETER

[I] `obj` tstring class object
[I] `str` Address of string

RETURN VALUE

true : If the strings match.
false : If the strings do not match.

EXAMPLE

The following compares a string that the object `my_str` includes with the string *User ID : 1234*, and then prints the results to standard output:

```
stdstreamio sio;
tstring     my_str = "User ID : 1234";

if (my_str == "User ID : 1234") {
    sio.printf("Character string same.\n");
}
else {
    sio.printf("Character string different.\n");
}
```

Result of execution

```
Character string same.
```

9.4.5 !=

NAME

!= — Comparison of strings

SYNOPSIS

```
bool operator!=(const tstring &obj) const;
bool operator!=(const char *str) const;
```

DESCRIPTION

Compares a string with a string specified to the right (argument) of the operator to verify whether they differ.

PARAMETER

[I] `obj` tstring class object
[I] `str` Address of string

RETURN VALUE

true : If the strings differ.
false : If the strings match.

EXAMPLE

The following code compares a string that the object `my_str` includes with the string `c_sentence`, and then prints the results to standard output:

```
stdstreamio sio;
tstring     my_str      = "User ID : 1234";
const char  *c_sentence = "User name : SUZUKI";

if (my_str != c_sentence) {
    sio.printf("Character string different.\n");
}
else {
    sio.printf("Character string same.\n");
}
```

Result of execution
```
Character string different.
```

9.5 Member functions

General information

The `size_t` type handles numerical values as unsigned integers. Setting a negative value to a function with a `size_t` type argument will increase the likelihood of the program aborting. Ensure to avoid setting any negative values.

9.5.1 length()

NAME

length() — Length of string

SYNOPSIS

```
size_t length() const;
```

DESCRIPTION

Returns the length of a string (`'\0'` not included).

RETURN VALUE

A string length

EXAMPLE

The following codes prints to standard output the length of a string that the object `my_str` includes:

```
stdstreamio sio;
tstring     my_str = "User'sFile.txt";

sio.printf("%zu\n", my_str.length());
```

Result of execution
```
14
```

9.5.2 max_length()

NAME

max_length() — Maximum value for the length of string that an object can handle

SYNOPSIS

```
size_t max_length() const;
```

DESCRIPTION

Returns the maximum string length in fixed-length buffer mode. If not in fixed-length buffer mode returns 0.

9.5.3 cstr(), c_str()

NAME

cstr(), c_str() — Beginning address for a string (read-only)

SYNOPSIS

```
const char *cstr() const;
const char *c_str() const;
```

DESCRIPTION

Returns the beginning address for a string inside an object.

If the string inside an object gets modified the modified string address is then acquired (Refer to EXAMPLE 2).

RETURN VALUE

Beginning address for a string

EXAMPLE-1

The following code creates the object `my_str` in NULL-free mode, and then prints it to standard output in verifying that the storage buffer for the internal string immediately after the object is created is not NULL. The string *This is a pen.* is assigned to `my_str`, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str(true);

if (my_str.c_str() != NULL) {
    sio.printf("The address of the character string is not NULL.\n");
}
else {
    sio.printf("The address of the character string is NULL.\n");
}

my_str = "This is a pen.";
sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
The address of the character string is not NULL.
This is a pen.
```

EXAMPLE-2

With the following code if a string that the object `my_str` includes is modified printts the result to standard output in order to verify that the new string address has been acquired:

```
stdstreamio sio;
tstring     my_str = "JAXA";

sio.printf("%s\n", my_str.cstr());

my_str = "ISAS";

sio.printf("%s\n", my_str.cstr());
```

Result of execution
```
JAXA
ISAS
```

9.5.4 str_ptr(), str_ptr_cs()

NAME

str_ptr(), str_ptr_cs() — Beginning address for string

SYNOPSIS

```
char *str_ptr();  ..... 1
const char *str_ptr() const;  ..... 2
const char *str_ptr_cs() const;  ..... 3
```

DESCRIPTION

Returns the beginning address for a string that is managed by the object.

Member function 1 is used if you wish to directly write into a string buffer inside the object. The size of the internal buffer using the resize() member function (§9.5.22) etc. must be adjusted to the length of the string you wish to write.

Member functions 2 and 3 have the same behavior as the cstr() member function (§9.5.3).

With the str_ptr() member function whether member function 1 or member function 2 is used is automatically determined by the presence or absence of the "const" attribute for an object. Member function 1 is automatically selected if the object does not have a "const" attribute and member function 2 if it does.

RETURN VALUE

Beginning address of string

WARNING

Ensure to avoid use of this member function unless absolutely necessary.

9.5.5 cchr()

NAME

cchr() — Reading of characters in specified position

SYNOPSIS

```
int cchr( size_t pos ) const;
```

DESCRIPTION

Returns the characters in position `pos` in a string inside an object. Please note that the lead position in strings is 0.

PARAMETER

[I] `pos` Position in string
([I] : Input, [O] : Output)

RETURN VALUE

Character in specified position	:	Normal termination
Negative value (Error)	:	If `pos` has a value longer than the length of a string inside an object specified to it.

EXAMPLE

The following code assigns the string *User'sFile3.txt* to a string that the object `my_str` includes, and then prints the 10th character in `my_str` to standard output:

```
stdstreamio sio;
tstring     my_str = "User'sFile3.txt";
int         i_ret;

if ((i_ret = my_str.cchr(10)) < 0){
    Error handling
}
sio.printf("%c\n", i_ret);
```

Result of execution

```
3
```

9.5.6 at(), at_cs()

NAME

at(), at_cs() — Reference to characters in specified position

SYNOPSIS

```
unsigned char &at( size_t pos );  ..... 1
const unsigned char &at( size_t pos ) const;  ..... 2
const unsigned char &at_cs( size_t pos ) const;  ..... 3
```

DESCRIPTION

Returns a reference to characters in position `pos` in a string inside an object. Please note that the lead position in strings is always 0.

Member function 1 can be used to both read and write characters whereas member functions 2 and 3 are used in reading only.

With the at() member function whether member function 1 or member function 2 is used is automatically determined by the presence or absence of a "const" attribute for an object. Member function 1 is automatically selected if the object does not have a "const" attribute and member function 2 if it does.

With member function 1 if the operating mode for an object is normal mode or NULL-free mode the string length is adjusted to make it `pos+1` and reading and writing performed even if `pos` is longer than the string length specified.
The same thing also occurs in fixed-length buffer mode, but if you specify a value for `pos`is longer than the maximum string length set when the object is created an exception occurs.

If a `pos` value is specified that is longer than the string length for member functions 2 and 3 an exception will occur in all the operating modes.

PARAMETER

[I] `pos` Position in string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to characters in a specified position

EXCEPTION

If in fixed-length buffer mode `pos` has a longer value than the maximum string length specified for it (Member function 1).
If `pos` has a longer value than the string length specified for it (Member functions 2 and 3).

EXAMPLE

The following code reads sixth character in a string that object `my_str` includes, and then prints the result to standard output.
The character *X* is written into the ninth character position in `my_str`, and then prints the result to standard output:

```
stdstreamio   sio;
tstring       my_str = "abcdefgh";
unsigned char c_read;

c_read = my_str.at(6);
sio.printf("%c\n", c_read);

my_str.at(9) = 'X';
sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
g
abcdefgh␣X
```

(␣ refers to white space character.)

9.5.7 update_length()

NAME

update_length() — Update length information in object (only for fixed-length buffer mode)

SYNOPSIS

```
tstring &update_length();
```

DESCRIPTION

When fixed-length buffer mode, this member function finds the terminating `'\0'` character in internal buffer, and updates internal information of object.

Because object of fixed-length mode manages both buffer length and length of string, `update_length()` should be used when characters are directly written to internal buffer of fixed-length mode.

9.5.8 dprint()

NAME

dprint() — Outputs object information to standard error output (For use in debugging)

SYNOPSIS

```
void dprint() const;
```

DESCRIPTION

Outputs information on an object to standard error output.

Member function designed for use in debugging user programs.

EXAMPLE

The following code outputs information on the object `my_str` to standard error output. The address for the object can be seen to be displayed in `[]`, which depends on the execution environment:

```
tstring my_str = "X68000 PRO";
my_str.dprint();
```

Result of execution

```
sli::tstring[obj=0x7fbffff640] = "X68000 PRO"
```

9.5.9 getstr()

NAME

getstr() — Copies (sub)string to an external buffer

SYNOPSIS

```
ssize_t getstr( char *dest_str, size_t buf_size ) const;  ..................  1
ssize_t getstr( size_t pos, char *dest_str, size_t buf_size ) const;  .....  2
```

DESCRIPTION

Copies a string inside an object to external buffer `dest_str`.

The size of `dest_str` is specified using `buf_size`. The number of characters written if sufficient buffer exists for `dest_str` is returned as the return value. A return value being larger than the size of `dest_str` therefore means that the buffer was insufficient.

Member function 1 copies an object to `dest_str`.

Member function 2 copies characters starting from position `pos` in a string inside an object to `dest_str`. Please note that the lead position in strings is always 0.

If the size of a buffer is insufficient for a string inside an object the string to which `dest_str` refers is to still terminates at `'\0'`. In this case member function 1 copies `buf_size-1`characters from the beginning of the string inside an object, whereas member function 2 copies `buf_size-1` characters from position `pos` in the string inside an object.

PARAMETER

[O]	`dest_str`	Address for external buffer to copy to
[I]	`buf_size`	Size of external buffer
[I]	`pos`	Position to start copying from

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Number of characters that can be copied if there is sufficient buffer length (`'\0'` not included).
Negative value (Error)	:	If `NULL` is set to `dest_str`, and a value other than 0 is set to `buf_size`.
	:	If `pos` has a value larger than the length of a string inside the object specified to it.

EXAMPLE

The following code copies characters from the beginning of the string that object `my_str`includes to external buffer `c_sentence`, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";
char        c_sentence[10] ;
int         i_ret  = 0;

if ((i_ret = my_str.getstr(0, c_sentence, sizeof(c_sentence))) < 0 ) {
    Error handling
}
else if ( sizeof(c_sentence) < i_ret ) {
    sio.printf("The length of buffer is insufficient. \n");
}
else {
    sio.printf("%s\n", c_sentence);
}
```

Result of execution

```
JAXA/ISAS
```

9.5.10 copy()

NAME

copy() — Copies (sub)string to an external object

SYNOPSIS

```
ssize_t copy( tstring *dest ) const;  ..... 1
ssize_t copy( size_t pos, tstring *dest ) const;  ..... 2
ssize_t copy( size_t pos, size_t n, tstring *dest ) const;  ..... 3
```

DESCRIPTION

Copies all or part of a string inside an object to external buffer `dest`.

If the operating mode used with `dest` is the fixed-length buffer mode strings are copied within the range of the maximum string length of `dest`. Please note the return value in this case is the number of characters returned that is written if there is a sufficient buffer for `dest`. A return value being larger than the size of the buffer for `dest` therefore means that the buffer was insufficient.

Member functions 1 copies objects to `dest`.

Member functions 2 and 3 copy a string starting from position `pos` in a string inside an object. Please note that the lead position in strings is always 0. In addition, member functions 3 enables you to specify the length `n` of a string to copy.

Member function that corresponds to the substr() function in Perl and PHP.

PARAMETER

[I]	`pos`	Position in a string inside an object to be copied
[I]	`n`	Number of characters to be copied
[O]	`dest`	Object for the external tstring class to copy to

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Number of characters that can be copied if there is sufficient buffer.
Negative value (Error)	:	If `pos` has a value larger than the length of a string inside the object specified to it.
	:	If `dest` is NULL.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code copies characters from the beginning of a string that object `my_str`includes to external object `my_id`, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_id(4);
tstring     my_str   = "User ID : 1234";
int         i_ret    = 0;

if ((i_ret = my_str.copy(10, 4, &my_id)) < 0 ) {
    Error handling
}
else if ( 4 < i_ret ) {
    sio.printf("The length of buffer is insufficient. \n");
}
else {
    sio.printf("%s\n", my_id.cstr());
}
```

Result of execution

```
1234
```

9.5.11 swap()

NAME

swap() — Swaps objects

SYNOPSIS

```
tstring &swap( tstring &sobj );
```

DESCRIPTION

Swaps the content of object `sobj` with the content of itself. The operating mode (Refer to §9.1) is not swapped when this takes place (Refer to EXAMPLE).

PARAMETER

[I/O]	`sobj`	Object for tstring class to swap the content with

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory.

EXAMPLE

The following code creates objects `my_str1` and `my_str2` in normal mode along with also object `my_str3` in fixed-length buffer mode that then has the maximum string length of 4 specified to it. It then prints the result to standard output in order to verify that the strings these objects include have been swapped by `swap()`:

```
stdstreamio sio;
tstring     my_str1;
tstring     my_str2;
tstring     my_str3(4);

my_str1 = "ISS/Kibo";
my_str2 = "JAXA/ISAS";
my_str3 = "NASA";

my_str1.swap(my_str2);
sio.printf("%s\n", my_str1.c_str());
sio.printf("%s\n", my_str2.c_str());

my_str1.swap(my_str3);
sio.printf("%s\n", my_str1.c_str());
sio.printf("%s\n", my_str3.c_str());
```

Result of execution

```
JAXA/ISAS
ISS/Kibo
NASA
JAXA
```

9.5.12 init()

NAME

init() — Complete initialization of objects

SYNOPSIS

```
tstring &init();  ..... 1
tstring &init( const tstring &src );  ..... 2
```

DESCRIPTION

Initializes objects.

Member function 1 completely initializes objects. The operating mode being NULL-free mode or fixed-length buffer mode results in initialization of the string buffer with the memory area maintained as is. With normal mode the memory area allocated to the string buffer inside an object is entirely released, and hence execution of the `cstr()` member function (§9.5.3) returns `NULL`.

Member function 2 initializes objects with the content of `src`.

PARAMETER

[I] `src` Object for tstring class that has the string to be sourced

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory (Member function 2).

EXAMPLE-1

The following code standard-outputs the result of execution to verify that a string that the object `my_str` includes has been changed by `init()`:

```
stdstreamio   sio;
tstring       my_str      = "User'sFile01.txt";
const tstring my_sentence = "JaxaData.txt";

sio.printf("%s\n", my_str.cstr());

my_str.init(my_sentence);

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
User'sFile01.txt
JaxaData.txt
```

EXAMPLE-2

The following code prints the result of execution to standard output in order to verify that a string that the object `my_str` includes has been changed by `init()`:

```
stdstreamio sio;
tstring     my_str = "User'sFile01.txt";

sio.printf("%s\n",my_str.cstr());

my_str.init();

sio.printf("%s\n",my_str.cstr());
```

Result of execution

```
User'sFile01.txt
(null)
```

9.5.13 printf(), vprintf(), assign(), assignf(), vassignf()

NAME

printf(), vprintf(), assign(), assignf(), vassignf() — Initialization of objects

SYNOPSIS

```
tstring &printf( const char *format, ... );  ..... 1
tstring &vprintf( const char *format, va_list ap );  ..... 2
tstring &assign( int ch, size_t n );  ..... 3
tstring &assign( const char *str );  ..... 4
```

DESCRIPTION

Initializes a string inside an object with the string specified with the arguments.

Member functions 1, 2, 6, and 7 initialize objects using strings created according to `format`. Member functions 1 and 6 convert each element of data of a variable-length argument, whereas member functions 2 and 7 convert the list `ap` of variable-length arguments, each depending on the format specified in `format` however. For more information on `format` refer to the descriptions provided in §8.1.14.

Member function 3 initializes the buffer for a string inside an object with the `n` characters provided by `ch`.

Member functions 4 and 5 initialize the buffer for a string inside an object with string `str`. Member function 5 also enables you to specify the length `n` of `str` used in initialization to be specified. `n` being larger than the string length of `str` results in the entire string in `str` being the target of initialization.

Member functions 8 and 9 initialize objects using the string in and after position `pos2` in object `src`. Please note that the lead position in strings is always 0. Member function 8 can be used without having specified `pos2`. In that case, however, the function is processed as though 0 had been specified. Member function 9 enables the length `n2` of the string used in initialization to be specified.

PARAMETER

[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of the variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`
[I]	`ch`	Character to be sourced
[I]	`str`	String to be sourced
[I]	`n`	Number of `ch` or length of `str`
[I]	`src`	Object for tstring class that includes the string to be sourced
[I]	`pos2`	Starting position of the string in `src` (If a substring of `src` is assigned)
[I]	`n2`	Length of string to be written (If a substring of `src` is assigned)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted using the specified conversion format (Member functions 1, 2, 6, and 7).

EXAMPLE-1

The following code prints the result of execution to standard output in order to verify that the string the object `my_str` includes has been initialized by `printf()`:

```
stdstreamio sio;
tstring     my_str;
```

```
sio.printf("%s\n",my_str.cstr());

my_str.printf("%s", "TEST_OK");

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
(null)
TEST_OK
```

EXAMPLE-2

This code initializes `my_str` using the five characters that start from the eleventh character *e* in the string `my_sentence` of the string which the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio   sio;
tstring       my_str;
const tstring my_sentence = "This is an eraser.";

sio.printf("%s\n", my_str.cstr());

my_str.assign(my_sentence, 11, 5);

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
(null)
erase
```

9.5.14 implode()

NAME

implode() — Sets a string that elements of a string array joined with delimiter

SYNOPSIS

```
tstring &implode( const char *const *arr, const char *delim );
```

DESCRIPTION

`implode()` joins all elements of a string array specified by argument `arr` with delimiter string `delim`, and store the result into the object.

This member function does not change object when `arr` is NULL.

PARAMETER

[I]	`arr`	String array (pointer array with NULL termination)
[I]	`delim`	Delimiter string

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

This code prepares a string array `arr`, joins all elements of it with delimiter `","`, and then prints the result to standard output:

```
stdstreamio sio;
const char *arr[] = {"X1", "MZ2861", "X68k", NULL};
tstring my_str;

my_str.implode(arr, ",");

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
X1,MZ2861,X68k
```

9.5.15 import_binary()

NAME

import_binary() — Import of binary data

SYNOPSIS

```
tstring &import_binary( const char *buf, size_t bufsize, int altchr = '\0' );
```

DESCRIPTION

This member function calls `this->resize(bufsize)`, and then reads `bufsize` bytes from the buffer specified by `buf`, and then stores it in an object. If `altchr` has anything else but `'\0'` specified and the buffer includes the character `'\0'` in it, then replaces the character with `altchr` before storing the buffer.

If `altchr` is omitted the character `'\0'` inside the buffer can be stored as is, but the member functions that search strings and perform pattern matching will not necessarily operate properly.

PARAMETER

[I]	`buf`	Address for user buffer
[I]	`bufsize`	Size of user buffer
[I]	`altchr`	Character that replaces the character `'\0'` if the character exists in the user buffer

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure a buffer inside an object.

9.5.16 put(), putf(), vputf()

NAME

put(), putf(), vputf() — Sets characters or strings to a given position

SYNOPSIS

```
tstring &put( size_t pos1, int ch, size_t n );  ..............................  1
tstring &put( size_t pos1, const char *str );  ...............................  2
tstring &put( size_t pos1, const char *str, size_t n );  .....................  3
tstring &putf( size_t pos1, const char *format, ... );  ......................  4
tstring &vputf( size_t pos1, const char *format, va_list ap );  ..............  5
tstring &put( size_t pos1, const tstring &src, size_t pos2 = 0 );  ...........  6
tstring &put( size_t pos1, const tstring &src, size_t pos2, size_t n2 );    7
```

DESCRIPTION

Writes to position `pos1` in a string inside an object with the string specified with the arguments. Please note that the lead position in strings inside an object and in the string specified is always 0.

`pos1` can take any given value. When the string buffer inside an object is smaller than that specified by the argument the size of the buffer gets automatically increased, with the added buffer being padded with the white space character `' '`, and then the string specified with the arguments written into the position of `pos1`. However, if the operating mode is fixed-length buffer mode the size does not get increased to any larger than the maximum string length set when the object is created. For example, if a `pos1` is specified that is longer than the maximum string length, a white space character is padded to the area from the end of the string to the full limit of the buffer size, with the characters and strings specified by the argument not being written.

Member function 1 writes `n` characters of `ch` to position `pos1` in the string inside an object.

Member functions 2 and 3 write string `str` to position `pos1` in the string inside an object. Member function 3 also enables length `n` of the `str` to be written to be specified. If `n` is larger than the length of string `str`, the whole string of `str` will be written.

Member functions 4 and 5 write strings that are created according to `format`. Member function 4 converts each element of data of a variable-length argument, whereas member function 5 converts list `ap` of variable-length arguments, each depending on the conversion specifications set in `format`. For more information on `format` refer to the descriptions provided in §8.1.14.

Member functions 6 and 7 write a string from position `pos2` in `src` to the string inside an object. Member function 6 can be used without specifying `pos2`. In that case, however, the function is processed as though 0 had been specified. Member function 7 enables the length `n2` of the `src` to be written to be specified.

PARAMETER

[I]	`pos1`	Position to start string inside an object
[I]	`ch`	Character to be sourced
[I]	`n`	Number of `ch` or length of `str`
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`
[I]	`src`	Object for tstring class that includes the string to be sourced
[I]	`pos2`	Starting position of the string in `src` (If substring of `src` is assigned)
[I]	`n2`	Length of string to be written (If substring of `src` is assigned)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted with the specified conversion format (Member functions 4 and 5).

EXAMPLE

The following code writes the four characters from the beginning of `my_sentence` to the sixth character position in the string that the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio   sio;
tstring       my_str      = "User'sFile01.txt";
const tstring my_sentence = "Data";

my_str.put(6, my_sentence, 0, 4);

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
User'sData01.txt
```

9.5.17 strcat(), strncat(), append(), appendf(), vappendf()

NAME

strcat(), strncat(), append(), appendf(), vappendf() — Addition of characters or strings

SYNOPSIS

```
tstring &strcat( const char *str );  ..... 1
tstring &strncat( const char *str, size_t n );  ..... 2
tstring &strcat( const tstring &src, size_t pos2 = 0 );  ..... 3
tstring &strncat( const tstring &src, size_t pos2, size_t n2 );  ..... 4
tstring &append( int ch, size_t n );  ..... 5
tstring &append( const char *str );  ..... 6
tstring &append( const char *str, size_t n);  ..... 7
tstring &appendf( const char *format, ... );  ..... 8
tstring &vappendf( const char *format, va_list ap );  ..... 9
tstring &append( const tstring &src, size_t pos2 = 0 );  ..... 10
tstring &append( const tstring &src, size_t pos2, size_t n2 );  ..... 11
```

DESCRIPTION

Adds the string specified with the arguments to a string inside an object.

The operating mode being fixed-length buffer mode results in the strings not being increased to any longer than the maximum string length set when the object was created. The string reaching the maximum string length or reaching the maximum string length while being added results in no further processing being performed.

Member functions 1, 2, 6 and 7 add the string `str` to the end of a string inside an object. In addition, member function 2 and 7 enable the length `n` of the `str` to be added to be specified.

`n` being larger than the string length of `str` results in the entire string being the target of addition.

Member functions 3, 4, 10, and 11 add a string in and after position `pos2` in string `src` to the end of a string inside the object. Please note that the lead position in strings is always 0. Member function 3 and 10 can be used without specifying `pos2`. In that case, however, the functions are processed as though 0 was specified. Member functions 4 and 11 enable the length `n2` of the `src` to be added to be specified.

Member function 5 adds `n` characters of `ch` to the end of a string inside an object.

Member functions 8 and 9 add strings created according to `format`. Member function 8 converts each element of data of a variable-length argument, whereas member function 9 converts the list `ap` of variable-length arguments, each depending on the conversion specifications set in `format`. For more information on `format` refer to the descriptions provided in §8.1.14.

PARAMETER

[I]	`str`	String to be sourced
[I]	`n`	Number of `ch` or length of `str`
[I]	`src`	Object for tstring class that includes string to be sourced
[I]	`pos2`	Starting position of the string in `src` (If substring of `src` is added)
[I]	`n2`	Length of string to be added (If substring of `src` is added)
[I]	`ch`	Character to be sourced
[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted with the specified conversion format (Member functions 8 and 9).

EXAMPLE

The following code adds the content of `my_suffix` to a string that the object `my_str`includes, and then prints the result to standard output:

```
stdstreamio   sio;
tstring       my_str    = "200X1231";
const tstring my_suffix = ".txt";

my_str.append(my_suffix);
sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
200X1231.txt
```

9.5.18 insert(), insertf(), vinsertf()

NAME

insert(), insertf(), vinsertf() — Insertion of characters or strings

SYNOPSIS

```
tstring &insert( size_t pos1, int ch, size_t n );  ..... 1
tstring &insert( size_t pos1, const char *str );  ..... 2
tstring &insert( size_t pos1, const char *str, size_t n );  ..... 3
tstring &insertf( size_t pos1, const char *format, ... );  ..... 4
tstring &vinsertf( size_t pos1, const char *format, va_list ap );  ..... 5
tstring &insert( size_t pos1, const tstring &src, size_t pos2 = 0 );  ..... 6
tstring &insert( size_t pos1, const tstring &src,size_t pos2, size_t n2 );  7
```

DESCRIPTION

Inserts the string specified with the arguments to position `pos1` in a string inside an object. Please note that the lead position in strings inside an object or in the string specified is always 0.

Member function 1 inserts `n` characters of `ch` to position `pos1` in the string inside an object.

Member functions 2 and 3 insert the string `str` to the position `pos1` in a string inside an object. Member function 3 also enables the length `n` of the `str` to be inserted to be specified.

Member functions 4 and 5 insert strings created according to `format`. Member function 4 converts each element of data of a variable-length argument, whereas member function 5 converts the list `ap` of variable-length arguments, each depending on the conversion specifications set in `format`. For more information on `formatt` refer to the descriptions provided in §8.1.14.

Member functions 6 and 7 insert a string in and after position `pos2` in the string `src` that is added to position `pos1` in the string inside an object. Member function 6 can be used without specifying `pos2`. In that case, however, the function is processed as though 0 was specified. Member function 7 enables the length `n2` of the `src` to be inserted to be specified.

PARAMETER

[I]	`pos1`	Position to start string inside object
[I]	`ch`	Character to be sourced
[I]	`n`	Number of `ch` or length of `str`
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`
[I]	`src`	Object for tstring class that includes string to be sourced
[I]	`pos2`	Starting position of the string inside `src` (If substring of `src` is assigned)
[I]	`n2`	Length of string to be inserted (If substring of `src` is assigned)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted with the specified conversion format (Member functions 4 and 5).

EXAMPLE

The following code inserts characters to a string that the object `my_str` includes according to format, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "123";

sio.printf("%s\n", my_str.cstr());

my_str.insertf(1,"%c",'+');

sio.printf("%s\n", my_str.cstr());

my_str.insertf(3,"%c",'=');

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
123
1+23
1+2=3
```

9.5.19 replace(), replacef(), vreplacef()

NAME

replace(), replacef(), vreplacef() — Replacement of strings

SYNOPSIS

```
tstring &replace( size_t pos1, size_t n1, int ch, size_t n2 );  ..... 1
tstring &replace( size_t pos1, size_t n1, const char *str );  ..... 2
tstring &replace( size_t pos1, size_t n1, const char *str, size_t n2 );  . 3
tstring &replacef( size_t pos1, size_t n1, const char *format, ... );  .... 4
tstring &vreplacef( size_t pos1, size_t n1,
                    const char *format, va_list ap );  ..... 5
tstring &replace( size_t pos1, size_t n1,
                  const tstring &src, size_t pos2 = 0 );  ..... 6
tstring &replace( size_t pos1, size_t n1, const tstring &src,
                  size_t pos2, size_t n2 );  ..... 7
```

DESCRIPTION

Replaces `n1` characters from position `pos1` in a string inside an object with the specified string. Please note that the lead position in strings inside an object and in the string specified is always 0.

`pos1` having a value larger than the length of the string inside an object specified to it results in the same processing as the `stacat()` member function (§9.5.17). The sum of `pos1` and `n1` being larger than the length of the string inside, an object or the string needing to be expanded or contracted because of the size difference with `n1` and `n2` results in the string being automatically adjusted. However, the operating mode being fixed-length buffer mode results in the strings not being expanded to any longer than the maximum string length set when the object was created.

Member function 1 replaces `n1` characters from the position of `pos1` in a string inside an object with `n2` characters of `ch`.

Member function 2 and 3 replace `n1` characters from position `pos1` in a string inside an object with the string `str`. Member function 3 also enables the length `n2` of the `str` to be replaced with to be specified.

Member functions 4 and 5 perform conversions using strings created according to `format`. Member function 4 converts each element of data of a variable-length argument, whereas member function 5 converts the list `ap` of variable-length arguments, each depending on the conversion specification set in `format`. For more information on `format` refer to the descriptions provided in §8.1.14.

Member functions 6 and 7 convert `n1` characters from position `pos1` in a string inside an object with a string from position `pos2` in `src`. Member function 6 can be used without specifying `pos2`. In that case, however, the function is processed as though 0 was specified. Member function 7 enables the length `n2` of the `src` to be replaced with to be specified.

PARAMETER

[I]	`pos1`	Position to start string inside object
[I]	`n1`	Number of characters to be replaced
[I]	`ch`	Character to be sourced
[I]	`n2`	Number of `ch` or length of string in `str` or `src`
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of a variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`
[I]	`pos2`	Starting position of the string inside `src` (If substring of `src` is assigned)
[I]	`src`	Object for tstring class that includes string to be sourced

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted with the specified conversion format (Member functions 4 and 5).

EXAMPLE-1

The following code replaces eight characters from the eighth character Y in a string that object `my_str` includes according to the specified format, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "UserNameYYYYMMDD.txt";
time_t      jikoku;
struct      tm  *lt;

time(&jikoku);
lt = localtime(&jikoku);

my_str.replacef(8, 8, "%d%d%d", 1900+lt->tm_year, lt->tm_mon, lt->tm_mday);

sio.printf("%s\n", my_str.cstr());
```

Result of execution
`UserName2009219.txt`

EXAMPLE-2
The following code replaces two characters from the eleventh character *2* in a string that the object `my_str` includes with 4 *X* characters, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "User ID : 1234";
int         i_ch   = 'X';

my_str.replace(11, 2, i_ch, 4);

sio.printf("%s\n", my_str.cstr());
```

Result of execution
`User ID : 1XXXX4`

EXAMPLE-3
The following code replaces five characters from the eleventh character *S* in a string that the object `my_str` includes with the string *Akar*, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "My name is Suzuki.";

my_str.replace(11, 5, "Akar", 4);

sio.printf("%s\n", my_str.cstr());
```

Result of execution
`My name is Akari.`

9.5.20 erase()

NAME
erase() — and then prints the result to standard output:

SYNOPSIS
`tstring &erase();` 1
`tstring &erase( size_t pos, size_t n = 1 );` 2

DESCRIPTION
Erases characters in a string inside an object.

Member function 1 erases all the characters (String length becomes zero).

Member function 2 erases `n` characters from position `pos`. Please note that the lead position in strings is always 0. If `n` is not specified one character is erased.

PARAMETER
[I] `pos` Position to start erasing
[I] `n` Number of characters to be erased
([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXCEPTION
If the system failed to secure an internal buffer.

EXAMPLE
The following code erases the first character in a string that the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str(7);

my_str.init("sibuki");

sio.printf("%s\n", my_str.cstr());

my_str.erase(0);

sio.printf("%s\n", my_str.cstr());
```

Result of execution
```
sibuki
ibuki
```

9.5.21 clean()

NAME
clean() — Pads existing entire strings with a given character

SYNOPSIS
```
tstring &clean( int ch = ' ' );
```

DESCRIPTION
Pads the entire string inside an object with the character `ch`. The function can also be used without specifying `ch`. In that case, however, the function is processed as though the white space character `' '` was specified. Executing `clean()` does not change the length of strings.

PARAMETER
[I] `ch` Character to pad a string with
([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXCEPTION
If the system failed to secure an internal buffer.

EXAMPLE
The following code pads a string that the object `my_str` includes with the character *, and then standard-outputs the result:

```
stdstreamio sio;
tstring     my_str = "Akari20060222.txt";

my_str.clean('*');
sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
*****************
```

9.5.22 resize()

NAME

resize() — Changes the length of strings

SYNOPSIS

```
tstring &resize( size_t len );
```

DESCRIPTION

Changes the length of a string inside an object to `len`.

If the string length is increased a string comprised of the white space character `' '` is added.

If the string length is contracted the string after `len` is deleted.

PARAMETER

[I] `len` String length after being changed
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code assigns a string of eight characters to a string that the object `my_str`includes, changes the string length to 3, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str;

my_str = "USR TEST";
sio.printf("%s\n", my_str.cstr());

my_str.resize(3);

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
USR TEST
USR
```

9.5.23 resizeby()

NAME

resizeby() — Changes the relative length of strings

SYNOPSIS

```
tstring &resizeby( ssize_t len );
```

DESCRIPTION

Changes the length of a string inside an object by the length in `len`.

If the string length is increased a string comprised of the white space character `' '` is added.

If the string length is contracted the string of the last abs(`len`) characters is deleted.

PARAMETER

[I] `len` Increase/decrease in string length
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

9.5.24 crop()

NAME

crop() — Cropping of strings

SYNOPSIS

```
tstring &crop( size_t pos );  ..................................................  1
tstring &crop( size_t pos, size_t n );  ........................................  2
```

DESCRIPTION

Crops a string inside an object into `n` characters from position `pos`. Please note that the lead position in strings is always 0.

Member function 1 crops characters from `pos` to the end of a string inside an object.

Member function 2 crops `n` characters from `pos` in a string inside an object. The sum of `pos` and `n` being larger than the string length results in characters being cropped from the string in and after `pos`.

If a value larger than the string length is specified to `pos` the string length becomes 0.

PARAMETER

[I] `pos` Position to start cropping
[I] `n` Number of characters to be cropped
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code crops eight characters from the fifth character *2* in a string that the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "Akari20060222.txt";

my_str.crop(5,8);
sio.printf("%s\n", my_str.cstr());
```

Result of execution

20060222

9.5.25 chomp()

NAME

chomp() — Elimination of newline characters

SYNOPSIS

```
tstring &chomp( const char *rs = "\n" );
tstring &chomp( const tstring &rs );
```

DESCRIPTION

Eliminates the newline character to the right of a string inside an object. Newline characters are specified by `rs`.

For example, with DOS-format text files "`str.chomp("\r\n");`" would need to be added, and with the supported UNIX, Mac and DOS formats "`str.chomp("\n").chomp("\r");`" would need to be added.

PARAMETER

[I] `rs` Newline string

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

9.5.26 trim()

NAME

trim() — Elimination of arbitrary characters at both ends of a string

SYNOPSIS

```
tstring &trim( int side_space );  ..... 1
tstring &trim( const char *side_spaces = " \t\n\r\f\v" );  ..... 2
tstring &trim( const tstring &side_spaces );  ..... 3
```

DESCRIPTION

Eliminates arbitrary characters at both ends of a string inside an object. Arbitrary characters are specified by the character `side_space` or string `side_spaces`.

Member function 1 eliminates `side_space` at both ends of a string inside an object.

Member function 2 can be used without specifying `side_spaces`. In that case, however, the white space character, horizontal tabulation character, newline character, carriage return character, file separator character or vertical tabulation character are eliminated as though `"␣\t\n\r\f\v"` had been specified (␣ refers to white space character).

Member functions 2 and 3 enable `side_spaces` to be specified as a simple list of characters, for example `" \t"` as well as values like `"[A-Z]"` or `"[^A-Z]"`, as used in regular expressions. In addition, the character classes provided in Table 19 can also be specified inside `"[ ...]"`.

Character class	Corresponding character	Corresponding function in `libc`
`[:alnum:]`	Alphabetical or numerical character	`isalnum()`
`[:alpha:]`	Alphabetical character	`isalpha()`
`[:cntrl:]`	Control character	`iscntrl()`
`[:digit:]`	Decimal number character	`isdigit()`
`[:graph:]`	Print character (White space character excluded)	`isgraph()`
`[:lower:]`	Lowercase alphabetical character	`islower()`
`[:print:]`	Print character for printing (White space character included)	`isprint()`
`[:punct:]`	Punctuation character	`ispunct()`
`[:space:]`	White space character	`isspace()`
`[:upper:]`	Uppercase alphabetical character	`isupper()`
`[:xdigit:]`	Hexadecimal number character	`isxdigit()`

Table 19: List of character classes available for use with [...].

For example, `"[[:digit:]]"` is equivalent to `"[0-9]"`. For the other possibilities refer to the functions manual for the corresponding functions in libc, as some of them depend on the locale.

Please note that you cannot use expressions like `"[0-9]abcdef"` but instead `"[0-9a-f]"` or `"0123456789abcdef"`. This then means that when `side_spaces` begins with `'['` it must end with `']'`.

PARAMETER

[I] `side_space` Arbitrary character
[I] `side_spaces` Arbitrary string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code eliminates from a string that the object `my_str` includes arbitrary characters at both ends of the string, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "\tThis is a pen. \n";

my_str.trim();
sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
This is a pen.
```

9.5.27 ltrim()

NAME

ltrim() — Elimination of space on left end of a string

SYNOPSIS

```
tstring &ltrim( int side_space );
tstring &ltrim( const char *side_spaces = " \t\n\r\f\v" );
tstring &ltrim( const tstring &side_spaces );
```

DESCRIPTION

Eliminates a white space character on the left end of a string inside an object. White space characters are specified by `side_space` or `side_spaces`.

`side_spaces` can be specified as a simple list of characters, for example `" \t"` as well as values like `"[A-Z]"` or `"[^A-Z]"`, as used in regular expressions. In addition, the character classes provided below can also be specified inside `"[ ...]"`:

`[:alnum:]`, `[:alpha:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]`, `[:xdigit:]`.

For example, `"[[:digit:]]"` is equivalent to `"[0-9]"`. For other possibilities refer to the manual for the `isalpha()` function in libc as some of them depend on the locale.

Please note that you cannot use expressions like `"[0-9]abcdef"`. but instead `"[0-9a-f]"` or `"0123456789abcdef"` This then means that when `side_spaces` begins with '`[`' it must end with '`]`'.

RETURN VALUE

Reference to itself

9.5.28 rtrim()

NAME

rtrim() — Elimination of space on the right end of a string

SYNOPSIS

```
tstring &rtrim( int side_space );
tstring &rtrim( const char *side_spaces = " \t\n\r\f\v" );
tstring &rtrim( const tstring &side_spaces );
```

DESCRIPTION

Eliminates white space character on the right end of a string inside an object. White space characters are specified by `side_space` or `side_spaces`.

`side_spaces` can be specified as a simple list of characters, for example `" \t"` as well as values like `"[A-Z]"` or `"[^A-Z]"`, as used in regular expressions. In addition, the character classes provided below can also be specified inside `"[ ...]"`:

`[:alnum:]`, `[:alpha:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]`, `[:xdigit:]`.

For example, `"[[:digit:]]"` is equivalent to `"[0-9]"`. For other possibilities refer to the manual for the isalpha() function in libc as some of them depend on the locale.

Please note that you cannot use expressions like `"[0-9]abcdef"` but instead `"[0-9a-f]"` or `"0123456789abcdef"`. This then means that when `side_spaces` begins with '`[`' it must end with '`]`'.

RETURN VALUE

Reference to itself

9.5.29 strreplace()

NAME

strreplace() — Search for and replace strings

SYNOPSIS

```
ssize_t strreplace( const char *org_str, const char *new_str, bool all = false );
ssize_t strreplace( size_t pos, const char *org_str, const char *new_str,
                    bool all = false );
ssize_t strreplace( const tstring &org_str, const char *new_str,
                    bool all = false );
ssize_t strreplace( size_t pos, const tstring &org_str, const char *new_str,
                    bool all = false );
ssize_t strreplace( const char *org_str, const tstring &new_str,
                    bool all = false );
ssize_t strreplace( size_t pos, const char *org_str, const tstring &new_str,
                    bool all = false );
ssize_t strreplace( const tstring &org_str, const tstring &new_str,
                    bool all = false );
ssize_t strreplace( size_t pos, const tstring &org_str, const tstring &new_str,
                    bool all = false );
```

DESCRIPTION

Searches for the string `org_str` from the left side of a string inside an object, and when found replaces it with the string `new_str`.

If a string is replaced the member functions return the position next to the string that was replaced. If that return value is then provided to `pos` the next part in which `org_str` is found can be replaced.

If the last argument `all` is `true` all the parts that match are replaced with the `new_str`.

If more advanced search processing is required replacement can be executed with extended regular expressions for the regreplace() member function (§9.5.30).

PARAMETER

[I]	`org_str`	String to be detected
[I]	`new_str`	String to be sourced for replacement
[I]	`pos`	Position to start string search
[I]	`all`	Replace all flags

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If the string specified is found the position next to the string replaced.
Negative value (Error)	:	If the character or string specified is not found.
	:	If there is no string inside an object.
	:	If `org_str` or `new_str` is NULL.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code replaces the host name part, `darts.isas.jaxa.jp`, of the URL string that the object `my_url` includes:

```
stdstreamio sio;
tstring my_url = "http://darts.isas.jaxa.jp/foo/";

my_url.strreplace("darts.isas.jaxa.jp", "darts.jaxa.jp");
sio.printf("my_url = %s\n", my_url.cstr());
```

Result of execution

```
http://darts.jaxa.jp/foo/
```

9.5.30 regreplace()

NAME

regreplace() — Replaces parts that match an extended regular expression

SYNOPSIS

```
ssize_t regreplace( const char *pat,
                    const char *new_str, bool all = false );  ..... 1
ssize_t regreplace( size_t pos, const char *pat,
                    const char *new_str, bool all = false );  ..... 2
ssize_t regreplace( const tstring &pat,
                    const char *new_str, bool all = false );  ..... 3
ssize_t regreplace( size_t pos, const tstring &pat,
                    const char *new_str, bool all = false );  ..... 4
ssize_t regreplace( const tregex &pat,
                    const char *new_str, bool all = false );  ..... 5
ssize_t regreplace( size_t pos, const tregex &pat,
                    const char *new_str, bool all = false );  ..... 6
ssize_t regreplace( const char *pat,
                    const tstring &new_str, bool all = false );  ..... 7
ssize_t regreplace( size_t pos, const char *pat,
                    const tstring &new_str, bool all = false );  ..... 8
ssize_t regreplace( const tstring &pat,
                    const tstring &new_str, bool all = false );  ..... 9
ssize_t regreplace( size_t pos, const tstring &pat,
                    const tstring &new_str, bool all = false );  ..... 10
ssize_t regreplace( const tregex &pat,
```

```
                    const tstring &new_str, bool all = false );  ..... 11
ssize_t regreplace( size_t pos, const tregex &pat,
                    const tstring &new_str, bool all = false );  ..... 12
```

DESCRIPTION

Replaces with the string `new_str` any parts of a string inside an object that match the POSIX extended regular expression (hereinafter referred to as regular expression) specified by `pat`. With a `new_str` back references `"\\0"` through `"\\9"` (`"\\0"` refers to the entire part that matches) can be used. If you want to provide the backslash itself specify `"\\\\"`.

The regreplace() member function updates the buffer inside an object that stores the result of regular expression matching. If the argument `all` is false information on the result can be acquired using `reg_elem_length()`, `reg_pos()`, `reg_length()`, `reg_cstr()` and `reg_cstrarray()`, The functions respectively return the number of elements in a result, the position of the matching string, the string length of the matching string, the string of characters in the matching string, and the pointer array for the string of characters in a string that matches. The prototypes for these member functions are as follows:

```
size_t reg_elem_length() const;
size_t reg_pos( size_t idx ) const;
size_t reg_length( size_t idx ) const;
const char *reg_cstr( size_t idx ) const;
const char *const *reg_cstrarray() const;
```

The element numbers starting from 0 are specified in the argument `idx`. In the 0th information on the entire matching string is stored, while in the first and later information on a substring that matches the regular expressions "`(...)`" is individually stored (i.e. back reference information). The return value for the reg_cstrarray() member function can also be assigned to an object for the tarray_tstring class (§10) using the `=` operator.

If the argument `all` is true information on the result cannot be acquired because the result of regular expression matching is reset.

With member functions 1 through 4 and 7 through 10 the regular expression `pat` is compiled, the result saved to the internal buffer that the functions themselves include, and matching then performed (If `pat`is the same as the one previously compiled it is not recompiled again).

With member functions 5, 6, 11 and 12 the object for the tregex class that holds the result of compiling the regular expression is specified. Regular expressions therefore need to be compiled in advance using the compile() member function of the tregex class before using the regmatch() member function (Refer to EXAMPLE 2 in §9.5.59).

In both cases if the regular expression fails to be compiled the content is output to the standard error output.

Attempts string matching from position `pos` in a string inside an object to the right. String matching is attempted within a range of up to where `'\0'` at the end of the string appears (Processing does not terminate when the newline character `'\n'` appears). If `pos` is not specified searches are made from the left end of a string inside an object. Please note that the lead position in strings is always 0.

If a string is replaced the member functions return the position of the string next to the string that was replaced. If this return value is then provided to `pos` the next part that matches can be replaced.

If the last argument `all` is `true` all the parts that match are replaced with `new_str`.

For more details on regular expressions refer to §9.5.59.

If you wish to execute replacement using a simpler search use of the strreplace() member function (§9.5.29) is recommended as it provides advantages in processing speed.

PARAMETER

[I]	`pat`	Character pattern (regular expression) or compiled object for the tregex class
[I]	`new_str`	String after being replaced
[I]	`pos`	Position to start string matching
[I]	`all`	Replace all flags

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Position next to string replaced
Negative value (Error)	:	If no string matches `pat`.
	:	If there is no string inside an object.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If `pat` or `new_str` is `NULL`.
	:	If the interval operators `{}` are not closed.
	:	If the list operators [] are not closed.
	:	If an unknown character class is set. ⟦For example use of *[:up:]*.⟧
	:	If a regular expression ends with a backslash.
	:	If the group operators () are not closed.
	:	If operators are used with an invalid range. ⟦For example use of *[9-0]*.⟧
	:	If there is an invalid back reference to the sub-expression `\(...\)`.
	:	If an invalid back reference operator is used.
	:	If invalid use of pattern operators such as a group or list is made. ⟦For example use of *[0-9)*.⟧
	:	If an invalid repetition operator is used in that `'*'` is the first character. ⟦For example use of *pat="*.txt"*.⟧

EXCEPTION

If the `regex` routine exhausted the memory space.
If the system failed to secure an internal buffer.
If the system encountered any corrupt memory.

EXAMPLE

The following code only replaces the host name part of the URL string that the object `my_url` includes:

```
stdstreamio sio;
tstring my_url = "http://darts.isas.jaxa.jp/foo/";

if ( my_url.regreplace("(http://)([^/]+)", "\\1darts.jaxa.jp") < 0 ) {
    Error handling
}
else {
    sio.printf("my_url = %s\n", my_url.cstr());
}
```

Result of execution
`http://darts.jaxa.jp/foo/`

9.5.31 tolower()

NAME
tolower() — Converts uppercase to lowercase characters

SYNOPSIS
`tstring &tolower( size_t pos = 0 );` 1
`tstring &tolower( size_t pos, size_t n );` 2

DESCRIPTION
Converts uppercase alphabetical characters in a string inside an object to lowercase characters. Please note that the lead position in strings is always 0.

Member function 1 processes from position `pos` in a string inside an object through to the right. The function can be used without specifying `pos`. In that case, however, the function is processed as though 0 was specified.

Member function 2 converts uppercase characters from `pos` to the `n`th character in a string inside an object to lowercase characters.

PARAMETER
[I] `pos` Position to start conversion
[I] `n` Number of characters to be converted
([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXAMPLE
The following code converts a string that the object `my_str` includes to lowercase, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";

my_str.tolower();

sio.printf("%s\n", my_str.cstr());
```

Result of execution
`jaxa/isas`

9.5.32 toupper()

NAME
toupper() — Converts lowercase to uppercase characters

SYNOPSIS
`tstring &toupper( size_t pos = 0 );` 1
`tstring &toupper( size_t pos, size_t n );` 2

DESCRIPTION

Converts lowercase alphabetical characters in a string inside an object to uppercase characters. Please note that the lead position in strings is always 0.

Member function 1 processes from position `pos` in a string inside an object through to the right. The function can be used without specifying `pos`. In that case, however, the function is processed as though 0 was specified.

Member function 2 converts lowercase characters from `pos` to the `n`th character in a string inside an object to uppercase characters.

PARAMETER

[I] `pos` Position to start conversion
[I] `n` Number of characters to be converted
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXAMPLE

The following code converts to uppercase characters four characters from the fifth character i in a string that the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "jaxa/isas";

my_str.toupper(5,4);

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
jaxa/ISAS
```

9.5.33 expand_tabs()

NAME

expand_tabs() — Replaces horizontal tabulation characters with white space characters

SYNOPSIS

```
tstring &expand_tabs( size_t tab_width = 8 );
```

DESCRIPTION

Replaces horizontal tabulation characters `'\t'` in a string inside an object with a white space character by tabulating the characters to the value in `tab_width`. Replacement is performed within the range of up to where `'\0'` at the end of the string appears (When the newline character `'\n'` appears the internal column numbers used for tabulation are reset, and processing continues to be performed). If characters do not need to be tabulated the tab characters can also be replaced using the `strreplace()` member function (§9.5.29) or the `regreplace()` member function (§9.5.30).

If `tab_width` is not specified or 0 is specified as `tab_width`, the function is processed as though 8 had been specified as `tab_width`.

If the change in `tab_width` increases the length of a string inside an object the size of the buffer gets automatically increased. However, the operating mode being fixed-length buffer

mode results in the string not being expanded to any longer than the maximum string length set when the object was created (Refer to EXAMPLE).

For example, with the string `"a\tbc\tdef` execution of `expand_tabs()` with `tab_width=3`results in the string being converted to `"a␣␣bc␣def"` (␣). In this example the first horizontal tabulation character is replaced by two white space characters because the sum of the number of characters in the string up to before the horizontal tabulation character 1 (`"a"`) and therefore the number of white space characters is tabulated to `tab_width`. In a similar manner, to tabulate to `tab_width` the sum of the number of characters in the string up to before the second horizontal tabulation character, which is 5 (`"a␣␣bc"`) and the number of white space characters, requires a white space character. For this reason the second horizontal tabulation character is replaced with a white space character.

PARAMETER

[I] `tab_width` A TAB width
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code creates the object `my_str1` in normal mode, with the object `my_str2` in fixed-length buffer mode that has the maximum string length of 11 being specified to it. The result is then printed to standard output in order to verify that the strings were adjusted with a TAB width when the horizontal tabulation characters inside the strings that the objects include were replaced using `expand_tabs()` with white space characters:

```
stdstreamio sio;
tstring     my_str1;
tstring     my_str2(11);

my_str1 = "Akari\tIbuki";
my_str2 = "Akari\tIbuki";

sio.printf("%s, %zu\n", my_str1.cstr(), my_str1.length());
sio.printf("%s, %zu\n", my_str2.cstr(), my_str2.length());

my_str1.expand_tabs();
my_str2.expand_tabs();

sio.printf("%s, %zu\n", my_str1.cstr(), my_str1.length());
sio.printf("%s, %zu\n", my_str2.cstr(), my_str2.length());
```

Result of execution

```
Akari Ibuki,11
Akari Ibuki,11
Akari   Ibuki,13(Akari␣␣␣Ibuki)
Akari   Ibu,11(Akari␣␣␣Ibu)
```

(␣ refers to a white space character.)

9.5.34 contract_spaces()

NAME

contract_spaces() — Replaces white space characters with TAB characters

SYNOPSIS

```
tstring &contract_spaces( size_t tab_width = 8 );
```

DESCRIPTION

Replaces with `'\t'` all occurrences of two or more contiguous white space characters `' '` in a string inside an object that tabulate to the specified TAB width of `tab_width`. However, replacement is performed within the range of up to where `'\0'` appears at the end of the string (When the newline character `'\n'` appears the internal column numbers used for tabulation are reset, and the processing continues). This member function performs the reverse operation to the operation described in §9.5.33. If characters do not need to be tabulated the white space characters can also be replaced using the `strreplace()` member function (§9.5.29) or the `regreplace()` member function (§9.5.30).

The function can be used without specifying `tab_width`. In that case, however, and when 0 is specified as `tab_width`, the function is processed as though 8 had been specified to `tab_width`.

For example, with the string `"abc␣␣␣␣␣␣␣␣de"` (␣ refers to s white space) executing `contract_spaces()` with `tab_width=4` results in the string being converted to `"abc␣\t␣␣␣de"`. In this example, to tabulate to `tab_width` the sum of the number of characters up to before the white space character of 3 (`"abc"`) and the number of white space characters requires a white space character. The first white space character is therefore not replaced, and remains as is. The four white space characters before replacement are then replaced with a horizontal tabulation character. Following those characters there are less white space characters than in `tab_width`, and hence the white space characters are not replaced with horizontal tabulation characters. The breakdown after conversion of the white space characters, which totaled 8 before replacement, reveals that white space characters + horizontal tabulation characters (for the four white space characters before replacement) + three white space characters (Refer to EXAMPLE).

PARAMETER

[I] `tab_width` A TAB width
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code prints the result to standard output in order to verify that the contiguous white space characters in the string that the object `my_str` includes were adjusted using `contract_spaces()` with a TAB width of 4:

```
stdstreamio sio;
tstring     my_str = "abc        de";

sio.printf("%s\n", my_str.cstr());
```

```
my_str.contract_spaces(4);

sio.printf("%s\n", my_str.cstr());
```

Result of execution

```
abc        de(abc␣␣␣␣␣␣␣␣de)
abc    de(abc␣\t␣␣␣de)
```

WARNING

Operation with `tab_width=1` cannot be defined.

9.5.35 atoi(), atol(), atoll()

NAME

atoi(), atol(), atoll() — Converts to integer value

SYNOPSIS

```
int atoi( size_t pos = 0 ) const;  ..... 1
int atoi( size_t pos, size_t n ) const;  ..... 2
long atol( size_t pos = 0 ) const;  ..... 3
long atol( size_t pos, size_t n ) const;  ..... 4
long long atoll( size_t pos = 0 ) const;  ..... 5
long long atoll( size_t pos, size_t n ) const;  ..... 6
```

DESCRIPTION

Converts characters in and after position `pos` in a string inside an object to decimal integer values. `atoi()` converts strings to an integer number of `int` type. In a similar manner `atol()` and `atoll()` convert strings to an integer number of `long` type and integer number of `long long` type, respectively. If a string inside an object includes any character other than `[0-9]` (excluding signs at the beginning) the characters after that character will not be processed. Please note that the lead position in strings is always 0.

Member functions 1, 3, 5 can be used without specifying the position `pos`. In that case, however, the functions are processed as though 0 was specified.

Member functions 2, 4, 6 convert `n` characters from position `pos` in a string inside an object to integer values.

PARAMETER

[I] `pos` Position in a string inside an object to be converted to integer value
[I] `n` Number of characters to be converted to integer values
([I] : Input, [O] : Output)

RETURN VALUE

Integer number : Integer value converted

EXCEPTION

If the system failed to secure an internal buffer (Member functions 2, 4, and 6).

EXAMPLE

The following code converts the second and following characters in a string that the object `my_str` includes to integer numbers of the `int` type, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "1234abc567";

sio.printf("%d\n", my_str.atoi(2));
```

Result of execution
34

WARNING
Once a character other than `[0-9]` appears conversion terminates. If you wish to verify whether all the characters have been converted, use the `strtol()` member function (§9.5.37) that includes the `endpos` argument.

9.5.36 atof()

NAME
atof() — Converts to real value

SYNOPSIS
```
double atof( size_t pos = 0 ) const;  ..... 1
double atof( size_t pos, size_t n ) const;  ..... 2
```

DESCRIPTION
Converts characters in and after position `pos` in a string inside an object to real values. If a string inside an object includes any character that cannot be handled as a real value the characters after that character will not be processed. Please note that the lead position in strings is always 0.

Strings that can be converted to real values include decimal numbers, hexadecimal numbers, infinity or `NAN` (an incalculable number). Decimal numbers consist of a decimal string of one or more characters, and can include a decimal point. The exponential part of a decimal number can consist of `'E'` or `'e'` with a positive or negative symbol (omissible) placed after it and followed by a decimal numerical string of one or more characters that reveal to what power of 10 the number is. The function also supports FORTRAN-format double-precision exponent representation (e.g., `1.2345D-10`) (Refer to EXAMPLE 2).

Hexadecimal numbers consist of `"0x"` or `"0X"` followed by a hexadecimal numerical string of one or more characters, and can include a decimal point. The binary exponential part can then be specified. The binary exponential part consists of `'P'` or `'p'` with a positive or negative symbol (omissible) placed after it followed by a decimal numerical string of one or more characters that reveal to what power of 2 the number is (Refer to EXAMPLE 3). Only a decimal point or a binary exponential can be used.

Infinity is referred to by `"INF"` or `"INFINITY"`, both being case-independent.

`NAN`s are referred to by `"NAN"` (case-independent), and may be followed by `'('`*string*`')'`.

Member function 1 can be used without specifying the position `pos`. In that case, however, the function is processed as though 0 was specified.

Member function 2 converts to a real value `n` characters from position `pos` in a string inside an object.

PARAMETER
[I] `pos` Position in a string inside an object to be converted to real value
[I] `n` Number of characters to be converted to real values
([I] : Input, [O] : Output)

RETURN VALUE

Real number : Value of `double` converted

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE-1

The following code converts the second and following characters in a string that the object `my_str` includes to a real number of the `double` type, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "1234abc567";

sio.printf("%f\n",  my_str.atof(2));
```

Result of execution

```
34.000000
```

EXAMPLE-2

The following code converts to a real number of the `double` type five characters from the first character of *2* in a string that the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "123D-456";

sio.printf("%f\n",  my_str.atof(1,5));
```

Result of execution

```
0.002300
```

EXAMPLE-3

The following code converts to a real number of the `double` type a string that the object `my_str` includes, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "0xabp2";

sio.printf("%f\n", my_str.atof());
```

Result of execution

```
684.000000
```

WARNING

Once an invalid number for the radix appears the conversion terminates. If you wish to verify whether all the characters were converted, use the `strtod()` member function (§9.5.39 that includes the `endpos` argument.

9.5.37 strtol(), strtoll()

NAME

strtol(), strtoll() — Converts to integer value

SYNOPSIS

```
long strtol( int base, size_t *endpos ) const;  ..............................  1
long strtol( size_t pos, int base, size_t *endpos ) const;  .................  2
long strtol( size_t pos, size_t n, int base, size_t *endpos ) const;  .....  3
long long strtoll( int base, size_t *endpos ) const;  ........................  4
long long strtoll( size_t pos, int base, size_t *endpos ) const;  .........  5
long long strtoll( size_t pos, size_t n, int base, size_t *endpos ) const;  6
```

DESCRIPTION

Converts a string inside an object to an integer value using the base number in `base`. `strtol()` converts strings to an integer number of the `long` type, with `strtoll()` similarly converting strings to an integer number of the `long long` type. Values of 2 to 36 or 0 can be specified in `base`. If 0 or 16 is specified the string can be prefixed with `'0x'`, and is then handled as a hexadecimal number. If `base` is 0 for any other strings than this the strings are handled as an octal number when they begin with `'0'`, or as a decimal number if otherwise (Refer to EXAMPLE-2). In addition, returns to `endpos` the position of any character that is not converted.

Member functions 1 and 4 convert a string inside an object to an integer number.

Member functions 2, 3, 5 and 6 convert characters from position `pos` in a string inside an object to an integer number. Please note that the lead position in strings is always 0. Member functions 3 and 6 also enable the length `n` of a string to be converted to an integer value to be specified.

PARAMETER

[I]	`pos`	Position in a string inside an object to be converted to an integer value
[I]	`n`	Number of characters to be converted to an integer value
[I]	`base`	Base number
[O]	`endpos`	Position of a character in a string inside an object that is not converted

([I] : Input, [O] : Output)

RETURN VALUE

Integer number : Integer value that is converted

EXCEPTION

If the system failed to secure an internal buffer (Member functions 3, and 6)

EXAMPLE-1

The following code converts a string that object `my_str` includes to an integer number of the `long` type, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "01234F57";
long        l_ret  = -1;
size_t      endpos = 0;

l_ret = my_str.strtol(0, 0, &endpos);
if (endpos == 0) {
```

```
    Error handling
}
else {
    sio.printf("%ld,%zu\n", l_ret, endpos);
}
```

Result of execution

```
668,5
```

EXAMPLE-2

The following code converts the second and later characters in a string that object `my_str` includes to an integer number of the `long` type, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "1234F57";
long        l_ret  = -1;
size_t      endpos = 0;

l_ret = my_str.strtol(2, 10, &endpos);
if ( endpos == 0 ) {
    Error handling
}
else {
    sio.printf("%ld, %zu\n", l_ret, endpos);
}
```

Result of execution

```
34,4
```

9.5.38 strtoul(), strtoull()

NAME

strtoul(), strtoull() — Converts to an unsigned integer value

SYNOPSIS

```
unsigned long strtoul( int base, size_t *endpos ) const;  ..................  1
unsigned long strtoul( size_t pos, int base, size_t *endpos ) const;  .....  2
unsigned long strtoul( size_t pos, size_t n,
                       int base, size_t *endpos ) const;  ..................  3
unsigned long long strtoull( int base, size_t *endpos ) const;  ............  4
unsigned long long strtoull( size_t pos, int base, size_t *endpos ) const;  5
unsigned long long strtoull( size_t pos, size_t n,
                             int base, size_t *endpos ) const;  ............  6
```

DESCRIPTION

Converts a string inside an object to an unsigned integer value using the base number in `base`. `strtoul()` converts strings to an integer number of the `unsigned long` type, while `strtoll()` converts strings to an integer number of the `unsigned long long` type. Values of 2 to 36 or 0 can be specified in `base`. If 0 or 16 is specified the string can be prefixed with `'0x'`, and is then handled as a hexadecimal number. If `base` is 0 for any other strings than this the strings are handled as an octal number when they begin with `'0'`, and as a decimal

number if otherwise. In addition, returns to `endpos` the position of any character that is not converted.

Member functions 1 and 4 convert a string inside an object to an unsigned integer number.

Member functions 2, 3, 5 and 6 convert characters from position `pos` in a string inside an object to an unsigned integer number. Please note that the lead position in strings is always 0. Member functions 3 and 6 also enable the length `n`of a string to be converted to an unsigned integer value to be specified.

PARAMETER

[I]	`pos`	Position in a string inside an object to be converted to an integer value
[I]	`n`	Number of characters to be converted to an integer value
[I]	`base`	Base number
[O]	`endpos`	Position of a character in a string inside an object that is not converted

([I] : Input, [O] : Output)

RETURN VALUE

Integer number : Integer value that is converted

EXCEPTION

If the system failed to secure an internal buffer (Member functions 3 and 6)

EXAMPLE

The following code converts a string that the object `my_str` includes to an integer number of the `unsigned long` type using a decimal number, and then prints the result to standard output:

```
stdstreamio   sio;
tstring       my_str = "-1abc";
unsigned long ul_ret = 0;
size_t        endpos = 0;

ul_ret = my_str.strtoul(10, &endpos);
if ( endpos == 0 ) {
    Error handling
}
else {
    sio.printf("%lu,%zu\n", ul_ret, endpos);
}
```

Result of execution

```
4294967295,2
```

9.5.39 strtod()

NAME

strtod() — Converts to real value

SYNOPSIS

```
double strtod( size_t *endpos ) const;  ..................................... 1
double strtod( size_t pos, size_t *endpos ) const;  ......................... 2
double strtod( size_t pos, size_t n, size_t *endpos ) const;  ............... 3
```

DESCRIPTION

Converts a string inside an object to a real value. In addition, returns to `endpos` the position in a string inside an object that is not converted.

The function also supports FORTRAN-format double-precision exponent representation (e.g., `1.2345D-10`)). For further information refer to the descriptions provided in §9.5.36 on the strings that can be converted.

Member function 1 converts a string inside an object to a real number.

Member functions 2 and 3 convert characters from position `pos` in a string inside an object to a real number. Please note that the lead position in strings is always 0. In addition, the member function enables the length `n` of a string to be converted to a real value to be specified.

PARAMETER

[I]	`pos`	Position in a string inside an object to be converted to a real value
[I]	`n`	Number of characters to be converted to a real value
[O]	`endpos`	Position of a character in a string inside an object that is not converted

([I] : Input, [O] : Output)

RETURN VALUE

Real number : The value for `double` that is converted

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE-1

The following code converts the string `-12.3D-4X56` that the object `my_str` includes to a real number of the `double` type, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "-12.3D-4X56";
double      d_ret  = 0;
size_t      endpos = 0;

d_ret = my_str.strtod(&endpos);
if ( endpos == 0 ) {
    Error handling
}
else {
    sio.printf("%f, %zu\n", d_ret, endpos);
}
```

Result of execution

```
-0.001230,8
```

EXAMPLE-2

The following code converts the string `infinity` that the object `my_str` includes to a real number of the `double` type, and then standard-outputs the result:

```
stdstreamio sio;
tstring     my_str = "infinity";
double      d_ret  = 0;
```

```
size_t      endpos = 0;

d_ret = my_str.strtod(&endpos);
if ( endpos == 0 ) {
    Error handling
}
else {
    sio.printf("%f, %zu\n", d_ret, endpos);
}
```

Result of execution

```
inf,8
```

9.5.40 scanf(), vscanf()

NAME

scanf(), vscanf() — Formatted input conversion

SYNOPSIS

```
int scanf( const char *format, ... ) const;
int vscanf( const char *format, va_list ap ) const;
```

DESCRIPTION

Reads a string inside an object according to the conversion specifications in `format`, and then stores it in the arguments after `format`.

The result of the conversion depending on the conversion specifications in `format` is read by `scanf()` to each element data of a variable-length argument, and by `vscanf()` to the list `ap` of variable-length arguments. For more on `format` refer to the descriptions provided in §8.1.11.

PARAMETER

[I]	`format`	Format specifications for reading
[O]	`...`	Each element data of a variable-length argument to which to write
[O]	`ap`	List of variable-length arguments to which to write

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Number of input elements successfully read and converted.
EOF (Error)	:	If an argument is insufficient or `format` is `NULL`.
	:	If there is no string inside an object.
	:	If the input converted to the integer type specified by `format` exceeds the size of the allowable storage of the appropriate integer type.

EXAMPLE

The following code converts a string that the object `my_str` includes to the string buffers `c_name` and `c_id`, according to the format *NAME ID : %9s %9s*. It then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str     = "NAME ID : SATO 1234";
char        c_name[10];
```

```
char        c_id[10];
int         i_ret      = 0;

if ((i_ret = my_str.scanf("NAME ID : %9s %9s", c_name,c_id)) == EOF) {
    Error handling
}
else {
    sio.printf("%s, %s, %d\n", c_name, c_id, i_ret);
}
```

Result of execution

```
SATO, 1234, 2
```

WARNING

The input when specifying "%s" in `format` of a string being larger than the size of the buffer used to store it will result in a buffer overrun occurring. For the method of avoiding this problem refer to the WARNING in §8.1.11.

9.5.41 strcmp(), compare()

NAME

strcmp(), compare() — Comparison of strings

SYNOPSIS

```
int strcmp( const char *str ) const;  ..... 1
int strcmp( size_t pos1, const char *str ) const;  ..... 2
int strcmp( const tstring &str, size_t pos2 = 0 ) const;  ..... 3
int strcmp( size_t pos1, const tstring &str, size_t pos2 = 0 ) const;  .... 4
int compare( const char *str ) const;  ..... 5
int compare( size_t pos1, const char *str ) const;  ..... 6
int compare( const tstring &str, size_t pos2 = 0 ) const;  ..... 7
int compare( size_t pos1, const tstring &str, size_t pos2 = 0 ) const;  .. 8
```

DESCRIPTION

`strcmp()` and `compare()` are member functions with different names that operate in the same manner.

The strcmp() member function and compare() member function both compare a string inside an object with the string `str` in a dictionary-like manner. The comparison is based on the character code for each of the characters in a string. Unlike the strncmp() member function (§9.5.42), however, these functions compare all the characters from the start position of a string.

The start position of a string inside an object is specified using `pos1`, while the position to start the external character string `str` is specified using `pos2`. Please note that the lead position in a string inside an object and in external character strings is always 0.

Member functions 3, 4, 7 and 8 can be used without specifying `pos2`. In that case, however, the functions are processed as though 0 was specified.

PARAMETER

[I] `str` String to be used in comparison
[I] `pos1` Position to start a string inside an object
[I] `pos2` Position to start a string in `str` (When comparing with a substring in `str`)
([I] : Input, [O] : Output)

RETURN VALUE

0	:	If a string inside an object is equal to `str`.
Positive value	:	If a string inside an object is larger in a dictionary-like manner than `str`.
Negative value	:	If a string inside an object is smaller in a dictionary-like manner than `str`.
256 (Error)	:	If a string inside an object has a buffer and `NULL` specified to `str`.
-256 (Error)	:	If a string inside an object does not have a buffer and `str` is specified.
	:	If `pos1` has a value larger than the length of a string inside an object specified to it.
	:	If `pos2` has a value larger than the string length of `str` specified to it.

EXAMPLE

The following code compares a string that the object `my_str` includes with the external character string *Akari20090303.txt*, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "Akari20060222.txt";

if (my_str.compare("Akari20090303.txt") == 0) {
    sio.printf("The same file\n");
}
else {
    sio.printf("Different file\n");
}
```

Result of execution

```
Different file
```

9.5.42 strncmp(), compare()

NAME

strncmp(), compare() — Partially compares strings

SYNOPSIS

```
int strncmp( const char *str, size_t n ) const;  ..... 1
int strncmp( size_t pos1, const char *str, size_t n ) const;  ..... 2
int strncmp( const tstring &str, size_t pos2, size_t n ) const;  ..... 3
int strncmp( size_t pos1, const tstring &str,
             size_t pos2, size_t n ) const;  ..... 4
int compare( const char *str, size_t n ) const;  ..... 5
int compare( size_t pos1, const char *str, size_t n ) const;  ..... 6
int compare( const tstring &str, size_t pos2, size_t n ) const;  ..... 7
int compare( size_t pos1, const tstring &str,
             size_t pos2, size_t n ) const;  ..... 8
```

DESCRIPTION

`strncmp()` and `compare()` are member functions with different names that operate in the same manner.

The strncmp() member function and the compare() member function compare a string inside an object with the string `str` in a dictionary-like manner. The comparison is based on the character code of each of the characters in a string.

The position to start a string inside an object is specified using `pos1`, while the position to start the external character string `str` is specified using `pos2`. Please note that the lead position in a string inside an object and in external character strings is always 0.

Unlike the strcmp() member function (§9.5.41) these functions compare the first `n` characters from the position to start a string.

PARAMETER

[I]	`str`	String to be used in comparison
[I]	`pos1`	Position to start a string inside an object
[I]	`pos2`	Position to start a string in `str` (When comparing with a substring in `str`)
[I]	`n`	Number of characters to be compared

([I] : Input, [O] : Output)

RETURN VALUE

0	:	If a string inside an object is equal to `str`.
Positive value	:	If a string inside an object is larger in a dictionary-like manner than `str`.
Negative value	:	If a string inside an object is smaller in a dictionary-like manner than `str`.
256 (Error)	:	If a string inside an object has a buffer and `NULL` specified to `str`.
-256 (Error)	:	If a string inside an object does not have a buffer and `str` is specified.
	:	If a string inside an object does not have a buffer and `n` is specified.
	:	If `pos1` has a value larger than the length of a string inside an object specified to it.
	:	If `pos2` has a value larger than the string length of `str` specified to it.

EXAMPLE

The following code compares eight characters from the fifth character of *2* in a string that the object `my_str` includes with the external character string *Akari20090303.txt*, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "Akari20060222.txt";

if (my_str.strncmp(5, "20060222", 8) == 0) {
    sio.printf("The same date\n");
}
else {
    sio.printf("Different date\n");
}
```

Result of execution

```
The same date
```

9.5.43 strcasecmp(), strncasecmp()

NAME

strcasecmp(), strncasecmp() — Comparison of strings (Case-independent)

SYNOPSIS

```
int strcasecmp( const char *str ) const;  ..................................... 1
int strcasecmp( size_t pos1, const char *str ) const;  ..................... 2
```

DESCRIPTION

Compares a string inside an object with the string `str` in a dictionary-like manner and without discriminating between uppercase and lowercase alphabetical characters. Processing results in both the string inside an object and the string `str` being converted to lowercase characters and then compared. The comparison after the strings are converted to lowercase characters is based on the character code of each of the characters inside the strings.

The position to start a string inside an object is specified using `pos1`, while the position to start the external character string `str` is specified using `pos2`. Please note that the lead position in a string inside an object and in external character strings is always 0.

`strcasecmp()` compares all the characters from the position to start a string, while `strncasecmp()` inquires whether the first `n` characters from the position to start a string match.

Member functions 3 and 4 can be used without specifying `pos2`. In that case, however, the functions are processed as though 0 was specified.

PARAMETER

[I]	`str`	String to be used for comparison
[I]	`pos1`	Position to start a string inside an object
[I]	`pos2`	Position to start a string in `str` (When comparing with a substring in `str`)
[I]	`n`	Number of characters to be compared

([I] : Input, [O] : Output)

RETURN VALUE

0	:	If a string inside an object is equal to `str`.
Positive value	:	If a string inside an object is larger in a dictionary-like manner than `str`.
Negative value	:	If a string inside an object is smaller in a dictionary-like manner than `str`.
256 (Error)	:	If a string inside an object has a buffer and `NULL` specified to `str`.
-256 (Error)	:	If a string inside an object does not have a buffer and `str` is specified.
	:	If a string inside an object does not have a buffer and `n` is specified (Member functions 5 to 8).
	:	If `pos1` has a value larger than the length of a string inside an object specified to it (Member functions 2, 4, 6, and 8).
	:	If `pos2` has a value larger than the string length of `str` specified to it (Member functions 3, 4, 7, and 8).

EXAMPLE

The following code compares a string that the object `my_str` includes with the external character string *suzuki* using `strcmp()` and `strcasecmp()`. It then prints the result to standard output in order to verify that the uppercase and lowercase alphabetical characters were not discriminated in `strcasecmp()`:

```
stdstreamio sio;
```

```
tstring     my_str = "SUZUKI";

if (my_str.strcmp("suzuki") == 0 ) {
    sio.printf("The same name\n");
} else {
    sio.printf("Different name\n");
}
if (my_str.strcasecmp("suzuki") == 0 ) {
    sio.printf("The same name\n");
} else {
    sio.printf("Different name\n");
}
```

Result of execution

```
Different name
The same name
```

9.5.44 isalpha(), isalnum(), isdigit(), islower(), isupper(), etc.

NAME

isalpha(), isalnum(), isdigit(), islower(), isupper(), etc. — Classification of characters

SYNOPSIS

```
bool isalnum( size_t pos ) const;
bool isalpha( size_t pos ) const;
bool iscntrl( size_t pos ) const;
bool isdigit( size_t pos ) const;
bool isgraph( size_t pos ) const;
bool islower( size_t pos ) const;
bool isprint( size_t pos ) const;
bool ispunct( size_t pos ) const;
bool isspace( size_t pos ) const;
bool isupper( size_t pos ) const;
bool isxdigit( size_t pos ) const;
```

DESCRIPTION

Classifies a character in position `pos` in a string inside an object according to the present locale. Please note that the lead position in strings is always 0. All the member functions correspond with the functions in libc. For the correspondence between these member functions and their characters refer to Table 18.

PARAMETER

[I] `pos` Position of character to be classified

([I] : Input, [O] : Output)

RETURN VALUE

true : If a character in `pos` matches a character that the member function corresponds to.

false : If a character in `pos` does not match a character that the member function corresponds to.

: If `pos` has a value larger than the length of a string inside an object specified to it.

EXAMPLE

The following code prints the result of execution to standard output in order to verify that a character is located fifth of the characters in a string that the object `my_str` includes is alphabetical or numerical.

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";

if ((my_str.isalnum(5)) == true ) {
    sio.printf("It is alphabetical or a figure.\n");
}
else {
    sio.printf("It is neither alphabetical nor a figure.\n");
}
```

Result of the execution

```
It is alphabetical or a figure.
```

9.5.45 strchr(), find()

NAME

strchr(), find() — Searches for characters from the left

SYNOPSIS

```
ssize_t strchr( int ch ) const;  ..... 1
ssize_t strchr( size_t pos, int ch ) const;  ..... 2
ssize_t strchr( size_t pos, int ch, size_t *nextpos ) const;  ..... 3
ssize_t find( int ch ) const;  ..... 4
ssize_t find( size_t pos, int ch ) const;  ..... 5
ssize_t find( size_t pos, int ch, size_t *nextpos ) const;  ..... 6
```

DESCRIPTION

`strchr()` and `find()` have different names but operate in the same manner.

Searches a string inside an object from the left to the right for the character `ch`, and then returns the position in which the character first appears.

`pos` being specified results in the search starting from position `pos` in a string inside an object. Please note that the lead position in strings is always 0.

If you wish to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With the variable referred to by `nextpos` when a character is found the position one character to the right of the position in which the character was found is returned, and if no character is found the length of the string itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`ch`	Character to be detected
[I]	`pos`	Position to start a string inside an object
[O]	`nextpos`	`pos` for the next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If the character specified is found the position of the beginning of the string.
Negative value (Error)	:	If the character specified is not found.
	:	If there is no string inside an object.
	:	If `pos` has a value larger than the length of a string inside an object (Member functions 2, 3, 5, and 6).

EXAMPLE

Refer to EXAMPLE in §9.5.46.

9.5.46 strstr(), find()

NAME

strstr(), find() — Searches for strings from the left

SYNOPSIS

```
ssize_t strstr( const char *str ) const;  ..................................... 1
ssize_t strstr( size_t pos, const char *str ) const;  ....................... 2
ssize_t strstr( size_t pos, const char *str, size_t *nextpos ) const;  .... 3
ssize_t strstr( const tstring &str ) const;  ................................ 4
ssize_t strstr( size_t pos, const tstring &str ) const;  .................... 5
ssize_t strstr( size_t pos, const tstring &str, size_t *nextpos ) const;   6
ssize_t find( const char *str ) const;  ..................................... 7
ssize_t find( const char *str, size_t n ) const;  ........................... 8
ssize_t find( size_t pos, const char *str ) const;  ......................... 9
ssize_t find( size_t pos, const char *str, size_t n ) const;  ............. 10
ssize_t find( size_t pos, const char *str, size_t *nextpos ) const;  ..... 11
ssize_t find( size_t pos, const char *str, size_t n,
              size_t *nextpos ) const;  ..................................... 12
ssize_t find( const tstring &str ) const;  ................................... 13
ssize_t find( size_t pos, const tstring &str ) const;  ...................... 14
ssize_t find( size_t pos, const tstring &str, size_t *nextpos ) const;  . 15
```

DESCRIPTION

`strstr()` and `find()` have different names but operate in the same manner.

Searches a string inside an object from the left of the string `str`, and then returns the position in which the string first appears.

If `pos` is specified the search starts from the position `pos` in a string inside an object. Please note that the lead position in strings is always 0.

If `n` is specified the function requests the position that matches the string `n` characters from the beginning of `str`.

If you wish to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. If a string that is one or more characters long is found the position of the same length as `str` to the right of the position in which the string is found is returned to the variable referred to by `nextpos`. If a string 0 characters long is found, and the position one character to the right of the position in which the string is found is equal or smaller than the string length for the function itself, that value is returned as the variable referred to by `nextpos`. In any other case than above the length of the string itself + 1 is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`pos`	Position to start a string inside an object
[I]	`str`	String to be detected
[I]	`n`	Number of characters to be detected
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a string specified is found the position of the beginning of the string.
Negative value (Error)	:	If a string specified is not found.
	:	If there is no string inside an object.
	:	If `pos` has a value larger than the length of a string inside an object.
	:	If `str` is `NULL` (Member functions 1, 2, 3, 7, 8, 9, 10, 11, and 12).

EXAMPLE

The following code searches a string that the object `my_str` includes from the left of the position of the string *ISAS*, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS/AKARI";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.strstr("ISAS")) < 0 ) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
5
```

9.5.47 strrchr(), rfind()

NAME

strrchr(), rfind() — Searches for characters from the right

SYNOPSIS

```
ssize_t strrchr( int ch ) const;  ..... 1
ssize_t strrchr( size_t pos, int ch ) const;  ..... 2
ssize_t strrchr( size_t pos, int ch, size_t *nextpos ) const;  ..... 3
ssize_t rfind( int ch ) const;  ..... 4
ssize_t rfind( size_t pos, int ch ) const;  ..... 5
ssize_t rfind( size_t pos, int ch, size_t *nextpos ) const;  ..... 6
```

DESCRIPTION

`strrchr()` and `rfind()` have different names but operate in the same manner.

Searches a string inside an object from the right to the left for the character `ch`, and then returns the position in which the character first appears. The position will be position from the left of a string. Please note that the lead position in strings is always 0.

When `pos` is specified the search starts from the position `pos` in a string inside an object to the left.

If you wish to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With the variable referred to by `nextpos`, when a character is found when pos is 1 or more the position one character to the left of the position in which the character was found is returned, and otherwise the length of the string itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`ch`	Character to be detected
[I]	`pos`	Position to start a string inside an object
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a character specified is found the position of the beginning of the string.
Negative value (Error)	:	If a character specified is not found.
	:	If there is no string inside an object.
	:	If `pos` has a value larger than the length of a string inside an object specified to it (Member functions 2, 3, 5, and 6).

EXAMPLE

Refer to EXAMPLE in §9.5.48.

9.5.48 strrstr(), rfind()

NAME

strrstr(), rfind() — Searches for strings from the right

SYNOPSIS

```
ssize_t strrstr( const char *str ) const;  ..... 1
ssize_t strrstr( size_t pos, const char *str ) const;  ..... 2
ssize_t strrstr( size_t pos, const char *str, size_t *nextpos ) const;  .. 3
ssize_t strrstr( const tstring &str ) const;  ..... 4
ssize_t strrstr( size_t pos, const tstring &str ) const;  ..... 5
ssize_t strrstr( size_t pos, const tstring &str, size_t *nextpos ) const;  6
ssize_t rfind( const char *str ) const;  ..... 7
ssize_t rfind( const char *str, size_t n ) const;  ..... 8
ssize_t rfind( size_t pos, const char *str ) const;  ..... 9
ssize_t rfind( size_t pos, const char *str, size_t n ) const;  ..... 10
ssize_t rfind( size_t pos, const char *str, size_t *nextpos ) const;  .... 11
ssize_t rfind( size_t pos, const char *str, size_t n,
               size_t *nextpos ) const;  ..... 12
ssize_t rfind( const tstring &str ) const;  ..... 13
ssize_t rfind( size_t pos, const tstring &str ) const;  ..... 14
ssize_t rfind( size_t pos, const tstring &str, size_t *nextpos ) const;  15
```

DESCRIPTION

`strrstr()` and `rfind()` have different names but operate in the same manner.

Searches a string inside an object from the right to the left of string `str`, and then returns the position in which the string first appears. The position will be the position from the left of a string. Please note that the lead position in strings is always 0.

If `pos` is specified the search starts from the position `pos` in a string inside an object to the left.

If `n` is specified the function requests the position that matches the string `n` characters from the beginning of `str`.

If you wish to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. If a string that is one or more characters long is found the position the same length as `str` to the left of the position in which the string was found is returned to the variable referred to by `nextpos`. If a string that is 0 characters long is found and the position one character to the left of the position in which the string was found is not a negative number, that value is returned to the variable referred to by `nextpos`. In any other case than above the length of the string itself + 1 is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`pos`	Position to start a string inside an object.
[I]	`str`	String to be detected
[I]	`n`	Number of characters to be detected
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a string specified is found the position of the beginning of the string.
Negative value (Error)	:	If a string that is specified is not found.
	:	If there is no string inside an object.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If `str` is `NULL` (Member functions 1, 2, 3, 7, 8, 9, 10, 11, and 12).

EXAMPLE

The following code searches a string that the object `my_str` includes from the right of the position of the string *AS*, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS/NASA";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.rfind("AS")) < 0 ) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
11
```

9.5.49 find_first_of()

NAME

find_first_of() — Detects from the left the characters contained in a character set

SYNOPSIS

```
ssize_t find_first_of( const char *str ) const;  ..... 1
ssize_t find_first_of( const char *str, size_t n ) const;  ..... 2
ssize_t find_first_of( size_t pos, const char *str ) const;  ..... 3
ssize_t find_first_of( size_t pos, const char *str, size_t n ) const;  .... 4
ssize_t find_first_of( size_t pos, const char *str,
                       size_t *nextpos ) const;  ..... 5
ssize_t find_first_of( size_t pos, const char *str, size_t n,
                       size_t *nextpos ) const;  ..... 6
ssize_t find_first_of( const tstring &str ) const;  ..... 7
ssize_t find_first_of( size_t pos, const tstring &str ) const;  ..... 8
ssize_t find_first_of( size_t pos, const tstring &str,
                       size_t *nextpos ) const;  ..... 9
```

DESCRIPTION

The find_first_of() member function searches a string inside an object from the left to the right of the characters contained in the character set `str`, and then returns the position in which any of those characters first appears. Please note that the lead position in strings is always 0.

If `pos` is specified the search starts from the position `pos` in a string inside an object.

If `n` is specified `n` characters from the beginning of `str` is a set.

Character sets are sets of characters expressed as a character string in which the order of characters has no meaning, unlike character strings. For example, if the character set is `"ABC"` the function searches for characters that match `'A'`, `'B'` or `'C'`.

With `tstring.h` macros in Table 20 are defined. You will find it useful to set them in `str`.

Macro definition	Corresponding character set
`CSET_ALNUM`	"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
`CSET_ALPHA`	"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
`CSET_LOWER`	"abcdefghijklmnopqrstuvwxyz"
`CSET_UPPER`	"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
`CSET_DIGIT`	"0123456789"
`CSET_XDIGIT`	"0123456789abcdefABCDEF"

Table 20: Definition of macro constants for use in character sets.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With a variable referred to by `nextpos` if a character is found the position one character to the right of the position in which the character was found is returned, and if no character is found the string length of the function itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

Apart from the method of specifying the character sets the function operates in the same manner as the strpbrk() member function (§9.5.53) does. `strpbrk()` also enables use of expressions like `"[A-Z]"` in character sets, and hence you may want to consider using the function.

PARAMETER

[I]	`str`	Character set to be detected
[I]	`n`	Number of characters in the character set `str`
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	The position of the character specified or a character contained in the character set.
Negative value (Error)	:	If a character specified or characters contained in the character set are not found.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.
	:	If `str` is `NULL` (Member functions 1 to 6).

EXAMPLE-1

The following code searches a string that the object `my_str` includes from the left for the position in which any character contained in the character set `CSET_DIGIT` appears, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "Akari20090306.txt";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.find_first_of(CSET_DIGIT)) < 0 ) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
5
```

EXAMPLE-2

The following code searches a string that the object `my_str` includes from the left for the position in which any of the character of *I*, *S*, *A* or *S* appears, and then prints the result to standard output: (This result can be used to identify the difference in the EXAMPLE in §9.5.45.)

```
stdstreamio sio;
tstring my_str = "JAXA/ISAS/AKARI";
ssize_t t_ret  = 0;

if ((t_ret = my_str.find_first_of("ISAS")) < 0) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution
```
1
```

9.5.50 find_last_of()

NAME

find_last_of() — Detects from the right characters contained in a character set

SYNOPSIS

```
ssize_t find_last_of( const char *str ) const;  ..............................  1
ssize_t find_last_of( const char *str, size_t n ) const;  ...................  2
ssize_t find_last_of( size_t pos, const char *str ) const;  .................  3
ssize_t find_last_of( size_t pos, const char *str, size_t n ) const;  .....  4
ssize_t find_last_of( size_t pos, const char *str,
                      size_t *nextpos ) const;  ..............................  5
ssize_t find_last_of( size_t pos, const char *str, size_t n,
                      size_t *nextpos ) const;  ..............................  6
ssize_t find_last_of( const tstring &str ) const;  ..........................  7
ssize_t find_last_of( size_t pos, const tstring &str ) const;  .............  8
ssize_t find_last_of( size_t pos, const tstring &str,
                      size_t *nextpos ) const;  ..............................  9
```

DESCRIPTION

The find_last_of() member function searches a string inside an object from the right to the left for the character `ch` or the characters contained in the character set `str`, and returns the position in which any of those characters first appears. The position will be a position from the left of a string. Please note that the lead position in strings is always 0.

If `pos` is specified the search is performed from the position `pos` in a string inside an object to the left.

If `n` is specified `n` characters from the beginning of `str` is a set.

`tstring.h` defines the `CSET_ALNUM`, `CSET_ALPHA`, `CSET_LOWER`, `CSET_UPPER`, `CSET_DIGIT` and `CSET_XDIGIT` that can be used for character sets. For more details of those and an explanation about character sets refer to the descriptions provided in §9.5.49.

If you wish to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With a variable referred to by `nextpos` if a character is found when `pos` is 1 or more, the position one character to the left of the position in which the character was found is returned, and otherwise the length of the string itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

Apart from the method of specifying character sets the function operates in the same manner as the strrpbrk() member function (§9.5.54) does. `strrpbrk()` enables use of expressions like `"[A-Z]"` in character sets, and hence you may want to consider using the function.

PARAMETER

[I]	`str`	Character set to be detected
[I]	`n`	Number of characters in the character set `str`
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Position of the character specified or a character contained in a character set.
Negative value (Error)	:	If a character specified or characters contained in a character set are not found.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.
	:	If `str` is `NULL` (Member functions 1 to 6)

EXAMPLE-1

The following code searches a string that the object `my_str` includes from the right for the position in which the character of either *A*, *S* appears at the end, and then prints the result to standard output: (This result can be used to identify the difference in the EXAMPLE in §9.5.47.)

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS/NASA";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.find_last_of("AS")) < 0) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
13
```

EXAMPLE-2

The following code searches a string that the object `my_str` includes from the right for the position in which any character contained in the character set `CSET_DIGIT` appears, and then prints the result to standard output: (This result can be used to identify the difference in EXAMPLE 2 in §9.5.49.)

```
stdstreamio sio;
tstring my_str = "Akari20090306.txt";
ssize_t t_ret  = 0;

if ((t_ret = my_str.find_last_of(CSET_DIGIT)) < 0) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
12
```

9.5.51 find_first_not_of()

NAME

find_first_not_of() — Detects from the left the position of a character not contained in a character set

SYNOPSIS

```
ssize_t find_first_not_of( const char *str ) const;  ..... 1
ssize_t find_first_not_of( const char *str, size_t n ) const;  ..... 2
ssize_t find_first_not_of( size_t pos, const char *str ) const;  ..... 3
ssize_t find_first_not_of( size_t pos, const char *str, size_t n ) const;  4
ssize_t find_first_not_of( size_t pos, const char *str,
                           size_t *nextpos ) const;  ..... 5
ssize_t find_first_not_of( size_t pos, const char *str, size_t n,
                           size_t *nextpos ) const;  ..... 6
ssize_t find_first_not_of( const tstring &str ) const;  ..... 7
ssize_t find_first_not_of( size_t pos, const tstring &str ) const;  ..... 8
ssize_t find_first_not_of( size_t pos, const tstring &str,
                           size_t *nextpos ) const;  ..... 9
ssize_t find_first_not_of( int ch ) const;  ..... 10
ssize_t find_first_not_of( size_t pos, int ch ) const;  ..... 11
ssize_t find_first_not_of( size_t pos, int ch, size_t *nextpos ) const;  12
```

DESCRIPTION

The find_first_not_of() member function searches a string inside an object from the left to the right for the character other than `ch` or the characters not contained in the character set `str`, and then returns the position in which any of those characters first appears. Please note that the lead position in strings is always 0.

If `pos` is specified the search starts from the position `pos` in a string inside an object.

If `n` is specified `n` characters from the beginning of `str` is a set.

`tstring.h` defines the `CSET_ALNUM`, `CSET_ALPHA`, `CSET_LOWER`, `CSET_UPPER`, `CSET_DIGIT` and `CSET_XDIGIT` that can be used for character sets. For more details on those and an explanation about character sets refer to the descriptions provided in §9.5.49.

If you wish to continuously search for characters or strings `nextpos` can be set to acquire the value that should be provided to `pos` in the next iteration. With a variable referred to by `nextpos` if a character is found the position one character to the right of the position in which the character was found is returned, and if no character is found the string length of the function itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

A method of using the strpbrk() member function (§9.5.53) to specify a character set as expressions like `"[^A-Z]"` also exists. You may want to consider using the function.

PARAMETER

[I]	`ch`	Character to be excluded from detection
[I]	`str`	Character set
[I]	`n`	Number of characters in the character set `str`
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Position of a character other than specified or a character not contained in a character set.
Negative value (Error)	:	If a character other than specified or characters not contained in a character set are not found.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.
	:	If `str` is `NULL` (Member functions 1 to 6).

EXAMPLE

The following code searches a string that the object `my_str` includes from the left for the position in which a character which is not any of the characters *A*, *J* and *X* appears, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.find_first_not_of("AJX")) < 0){
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

4

9.5.52 find_last_not_of()

NAME

find_last_not_of() — Detects from the right the position of a character not contained in a character set

SYNOPSIS

```
ssize_t find_last_not_of( const char *str ) const;  ..... 1
ssize_t find_last_not_of( const char *str, size_t n ) const;  ..... 2
ssize_t find_last_not_of( size_t pos, const char *str ) const;  ..... 3
ssize_t find_last_not_of( size_t pos, const char *str, size_t n ) const;  4
ssize_t find_last_not_of( size_t pos, const char *str,
                          size_t *nextpos ) const;  ..... 5
ssize_t find_last_not_of( size_t pos, const char *str, size_t n,
                          size_t *nextpos ) const;  ..... 6
ssize_t find_last_not_of( const tstring &str ) const;  ..... 7
ssize_t find_last_not_of( size_t pos, const tstring &str ) const;  ..... 8
ssize_t find_last_not_of( size_t pos, const tstring &str,
                          size_t *nextpos ) const;  ..... 9
ssize_t find_last_not_of( int ch ) const;  ..... 10
ssize_t find_last_not_of( size_t pos, int ch ) const;  ..... 11
ssize_t find_last_not_of( size_t pos, int ch, size_t *nextpos ) const;  . 12
```

DESCRIPTION

find_last_not_of() member function searches a string inside an object from the right to the left for the character other than `ch` or the characters not contained in the character set `str`, and returns the position in which any of those characters first appears. The position will be a position from the left end of a string. Please note that the lead position in strings is always 0.

If `pos` is specified the search starts from the position `pos` in a string inside an object to the left.

If `n` is specified `n` characters from the beginning of `str` is a set.

`tstring.h` defines the `CSET_ALNUM`, `CSET_ALPHA`, `CSET_LOWER`, `CSET_UPPER`, `CSET_DIGIT` and `CSET_XDIGIT` that can be used for character sets. For more details on those and an explanation about character sets refer to the descriptions provided in §9.5.49.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With a variable referred to by `nextpos` if a character is found when `pos` is 1 or more the position one character to the left of the position in which the character was found is returned, and otherwise the string length of the function itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

A method of using the strrpbrk() member function (§9.5.54) to specify a character set as expressions like `"[^A-Z]"` also exists. You may want to consider using the function.

PARAMETER

[I]	`ch`	Character to be excluded from detection
[I]	`str`	Character set not included in a string
[I]	`n`	Number of characters in the character set `str`
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	The position of a character other than specified or a character not contained in a character set.
Negative value (Error)	:	If a character other than specified or characters not contained in a character set are not found.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.
	:	If `str` is `NULL` (Member functions 1 to 6).

EXAMPLE

The following code searches a string that the object `my_str` includes from the right for the position in which a character which is not any of the characters *N*, *A* and *S* appears, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.find_last_not_of("NASA",3)) < 0) {
    Error handling
```

```
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
5
```

9.5.53 strpbrk()

NAME

strpbrk() — Detects from the left characters contained in a character set

SYNOPSIS

```
ssize_t strpbrk( const char *accept ) const;  ..............................  1
ssize_t strpbrk( size_t pos, const char *accept ) const;  ..................  2
ssize_t strpbrk( size_t pos, const char *accept, size_t *nextpos ) const;  3
ssize_t strpbrk( const tstring &accept ) const;  ...........................  4
ssize_t strpbrk( size_t pos, const tstring &accept ) const;  ..............  5
ssize_t strpbrk( size_t pos, const tstring &accept,
                 size_t *nextpos ) const;  ..................................  6
```

DESCRIPTION

Searches a string inside an object from the left to the right for the characters contained in the character set `accept`, and returns the position in which any of those characters first appears. Please note that the lead position in strings is always 0.

If `pos` is specified the search starts from the position `pos` in a string inside an object.

In addition to the features of the `find_first_of()` member function (§9.5.49), these member functions enable `accept` to be specified as a simple list of characters like `"xyz"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions provided in §9.5.26.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With a variable referred to by `nextpos` if a character is found the position one character to the right of the position in which the character was found is returned, and if no character is found the string length of the function itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`accept`	Character set to be detected
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	The position of a character specified or a character contained in a character set.
Negative value (Error)	:	If a character specified or characters contained in a character set are not found.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.
	:	If `accept` is `NULL`.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code searches a string that the object `my_str` includes from the left for the position in which any character contained in the character set *[:digit:]* appears, and then prints the result to standard output: (This result can be used to verify that the same result as in EXAMPLE 1 in §9.5.49 can be obtained.)

```
stdstreamio sio;
tstring     my_str = "Akari20090306.txt";
ssize_t     t_ret  = 0;

if ((t_ret = my_str.strpbrk("[[:digit:]]")) < 0) {
    Error handling
}
else {
    sio.printf("%zd\n", t_ret);
}
```

Result of execution

```
5
```

9.5.54 strrpbrk()

NAME

strrpbrk() — Detects from the right characters contained in a character set

SYNOPSIS

```
ssize_t strrpbrk( const char *accept ) const;  ..............................  1
ssize_t strrpbrk( size_t pos, const char *accept ) const;  ..................  2
ssize_t strrpbrk( size_t pos, const char *accept, size_t *nextpos ) const;  3
ssize_t strrpbrk( const tstring &accept ) const;  ............................  4
ssize_t strrpbrk( size_t pos, const tstring &accept ) const;  ..............  5
ssize_t strrpbrk( size_t pos, const tstring &accept,
                  size_t *nextpos ) const;  ..................................  6
```

DESCRIPTION

Searches a string inside an object from the right to the left for the characters contained in the character set `accept`, and then returns the position in which any of those characters first appears. The position will be a position from the left end of a string. Please note that the lead position in strings is always 0.

If `pos` is specified the search starts from the position `pos` in a string inside an object.

In addition to the features of the `find_last_of()` member function (§9.5.50), these member functions enable `accept` to be specified as a simple list of characters like `"xyz"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions provided in §9.5.26.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. With a variable referred to by `nextpos` if a character is found when `pos` is 1 or more the position one character to the left of the position in which the character was found is returned, and otherwise the string length of the function itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`accept`	Character set to be detected
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for the next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	The position of a character specified or a character contained in a character set.
Negative value (Error)	:	If a character specified or characters contained in a character set are not found.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.
	:	If `accept` is `NULL`.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code searches the string `my_str` from the right for the parts with a number, and then lists any such parts found:

```
stdstreamio sio;
               /* 012345678 */
tstring my_str = "Z80A 4MHz";
size_t pos = my_str.length() - 1;
ssize_t fpos;
while ( 0 <= (fpos=my_str.strrpbrk(pos, "[0-9]", &pos)) ) {
    sio.printf("fpos = %zd  nextpos = %zu\n", fpos, pos);
}
```

Result

```
fpos = 5  nextpos = 4
fpos = 2  nextpos = 1
fpos = 1  nextpos = 0
```

Also refer to the EXAMPLE in §9.5.53.

9.5.55 strspn()

NAME

strspn() — Inquires from the left side the length of continuous characters contained in a character set

SYNOPSIS

```
size_t strspn( const char *accept ) const;  ..................................  1
size_t strspn( size_t pos, const char *accept ) const;  ....................  2
size_t strspn( size_t pos, const char *accept, size_t *nextpos ) const;  .  3
size_t strspn( const tstring &accept ) const;  ..............................  4
size_t strspn( size_t pos, const tstring &accept ) const;  ..................  5
size_t strspn( size_t pos, const tstring &accept, size_t *nextpos ) const;  6
size_t strspn( int accept ) const;  ..........................................  7
size_t strspn( size_t pos, int accept ) const;  ..............................  8
size_t strspn( size_t pos, int accept, size_t *nextpos ) const;  ...........  9
```

DESCRIPTION

Searches a string inside an object from the left to the right for the length in which the character set `accept` continues, and then returns the length.

If `pos` is specified the search starts from the position `pos` in a string inside an object. Please note that the lead position in strings is always 0.

Member functions 1 to 6 enable `accept` to be specified as a simple list of characters like `"xyz"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions provided in §9.5.26.

With member functions 1 to 6 if the character set `accept` is `NULL` all the characters will be targeted (Refer to EXAMPLE 3).

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. If the return value of this member function is 1 or more, the position as long as the return value to the right of `pos` is returned as the variable referred to by `nextpos`. If the return value for this member function is 0 and the position one character to the right of `pos` is smaller than the length of the string itself, that value is returned as the variable referred to by `nextpos`. In any other case than above the length of the string itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`accept`	Character set to be detected
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Positive value	:	The length of which characters contained in a character set continue.
0	:	If a character set does not continue from the position to start counting.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE-1

The following code searches a string that the object `my_str` includes from the left to identify how many consecutive characters consisting of any of the characters *A*, *J* and *X* there are, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";
size_t      t_ret  = 0;

t_ret = my_str.strspn("AJX");
if (t_ret == 0) {
    Error handling
}
else {
    sio.printf("%zu\n", t_ret);
}
```

Result of execution

```
4
```

EXAMPLE-2

The following code searches a string that the object `my_str` includes from the left to identify how many consecutive characters consisting of the characters contained in the character set *[:upper:]* there are, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";
size_t      t_ret  = 0;

t_ret = my_str.strspn("[[:upper:]]");
if (t_ret == 0) {
    Error handling
}
else {
    sio.printf("%zu\n", t_ret);
}
```

Result of execution

```
4
```

EXAMPLE-3

The following code performs a search for `NULL` character sets. It searches a string that the object `my_str` includes to standard output how many consecutive characters consisting of the characters contained in the character set there are from the second character in the string:

```
stdstreamio sio;
tstring     my_str = "JAXA/ISAS";
const char  *c_p   = NULL;
size_t      t_ret  = 0;

t_ret = my_str.strspn(2, c_p);
```

```
if (t_ret == 0) {
    Error handling
}
else {
    sio.printf("%zu\n", t_ret);
}
```

Result of execution
7

9.5.56 strrspn()

NAME

strrspn() — Identifies from the right the consecutive length of characters contained in a character set

SYNOPSIS

```
size_t strrspn( const char *accept ) const; ..... 1
size_t strrspn( size_t pos, const char *accept ) const; ..... 2
size_t strrspn( size_t pos, const char *accept, size_t *nextpos ) const; 3
size_t strrspn( const tstring &accept ) const; ..... 4
size_t strrspn( size_t pos, const tstring &accept ) const; ..... 5
size_t strrspn( size_t pos, const tstring &accept, size_t *nextpos ) const; 6
size_t strrspn( int accept ) const; ..... 7
size_t strrspn( size_t pos, int accept ) const; ..... 8
size_t strrspn( size_t pos, int accept, size_t *nextpos ) const; ..... 9
```

DESCRIPTION

Searches a string inside an object from the right to the left in identifying the consecutive length of the character set `accept`, and then returns the length value.

If `pos` is specified the search starts from the position `pos` in a string inside an object. Please note that the lead position in strings is always 0.

Member functions 1 to 6 enable `accept` to be specified as a simple list of characters like `"xyz"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions provided in §9.5.26.

With member functions 1 to 6 if the character set `accept` is `NULL` all the characters are targeted (Refer to EXAMPLE 3 in §9.5.55).

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided `pos` in the next iteration. If the return value for the member function is 1 or more the position the same length as the return value to the left of `pos` is returned as the variable referred to by `nextpos`. If the return value for this member function is 0 and the position one character to the left of `pos` is not a negative number that value is returned as the variable referred to by `nextpos`. In any other case than above the length of the string itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

PARAMETER

[I]	`accept`	Character set to be detected
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Positive value	:	The length of consecutive characters contained in a character set.
0	:	If a character set does not continue from the position to start counting.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code searches a string that the object `my_str` includes from the right for the parts in which a number continues, and then prints the result to standard output:

```
stdstreamio sio;
size_t pos, len;
               /* 012345678 */
tstring my_str = "Z80A 4MHz";

pos = my_str.length() - 1;
do {
    sio.printf("curpos = %zu  ", pos);
    len = my_str.strrspn(pos, "[0-9]", &pos);
    sio.printf("ret_len = %zu  nextpos = %zu\n", len, pos);
} while ( pos < my_str.length() );
```

Result of execution

```
curpos = 8  ret_len = 0  nextpos = 7
curpos = 7  ret_len = 0  nextpos = 6
curpos = 6  ret_len = 0  nextpos = 5
curpos = 5  ret_len = 1  nextpos = 4
curpos = 4  ret_len = 0  nextpos = 3
curpos = 3  ret_len = 0  nextpos = 2
curpos = 2  ret_len = 2  nextpos = 0
curpos = 0  ret_len = 0  nextpos = 9
```

9.5.57 strcspn()

NAME

strcspn() — Inquires from the left the length of consecutive characters not contained in a character set

SYNOPSIS

```
size_t strcspn( const char *reject ) const;  ..................................  1
size_t strcspn( size_t pos, const char *reject ) const;  ...................  2
size_t strcspn( size_t pos, const char *reject, size_t *nextpos ) const;   3
size_t strcspn( const tstring &reject ) const;  ..............................  4
size_t strcspn( size_t pos, const tstring &reject ) const;  .................  5
size_t strcspn( size_t pos, const tstring &reject, size_t *nextpos ) const;  6
size_t strcspn( int reject ) const;  .........................................  7
```

DESCRIPTION

Searches a string inside an object from the left to the right for the length of a consecutive string until the character set `reject` first appears, and then returns the length.

If `pos` is specified the search starts from the position pos in a string inside an object. Please note that the lead position in strings is always 0.

Member functions 1 to 6 enable `reject` to be specified as a simple list of characters like `"ijk"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions provided in §9.5.26.

With member functions 1 to 6 if the character set `reject` is `NULL` the target character set will be all the characters.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. If the return value for the member function is 1 or more the position the same length as the return value to the right of `pos` is returned as the variable referred to by `nextpos`. If the return value for this member function is 0 and the position one character to the right of `pos` is smaller than the length of the string itself that value is returned as the variable referred to by `nextpos`. In any other case than above the length of the string itself is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

A method of using the strspn() member function (§9.5.55) to specify a character set as an expression like `"[^A-Z]"` also exists, which may want to consider using.

PARAMETER

[I]	`reject`	Character set not to be detected
[I]	`pos`	Position to start detection
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Positive value	:	The consecutive length of characters not contained in a character set.
0	:	If characters not contained in a character set do not continue from the position to start counting.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If there is no string inside an object.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code searches a string that the object `my_str` includes from the left for the number of consecutive characters until the character *0* appears in the string, and then prints the result to standard output:

```
stdstreamio sio;
tstring     my_str = "Akari20090309.txt";
int         i_ch   = '0';
size_t      t_ret  = 0;
```

```
if ((t_ret = my_str.strcspn(i_ch)) == 0) {
    Error handling
}
else {
    sio.printf("%zu\n", t_ret);
}
```

Result of execution

```
6
```

9.5.58 strmatch(), fnmatch(), pnmatch()

NAME

strmatch(), fnmatch(), pnmatch() — Attempts Shell-like string matching

SYNOPSIS

```
int strmatch( const char *pat ) const;  ..... 1
int strmatch( size_t pos, const char *pat ) const;  ..... 2
int strmatch( const tstring &pat ) const;  ..... 3
int strmatch( size_t pos, const tstring &pat ) const;  ..... 4
int fnmatch( const char *pat ) const;  ..... 5
int fnmatch( size_t pos, const char *pat ) const;  ..... 6
int fnmatch( const tstring &pat ) const;  ..... 7
int fnmatch( size_t pos, const tstring &pat ) const;  ..... 8
int pnmatch( const char *pat ) const;  ..... 9
int pnmatch( size_t pos, const char *pat ) const;  ..... 10
int pnmatch( const tstring &pat ) const;  ..... 11
int pnmatch( size_t pos, const tstring &pat ) const;  ..... 12
```

DESCRIPTION

Attempts string matching on a string inside an object using Shell wild card patterns, and then returns the result.

`strmatch()` attempts matching with the character pattern `pat`, from the position `pos` in a string inside an object to the right. The string matching is attempted within the range of up to where `'\0'` appears at the end of the string (When the newline character `'\n'` does appear the processing still does not terminate). If `pos` is not specified searches are made from the left end of a string inside an object. Please note that the lead position in strings is always 0.

Wild cards available for `pat` can be specified using `'*'` and `'?'` and expressions like `"[A-Z]"` as in regular expressions[11]. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions provided in §9.5.26.

`fnmatch()` provides `strmatch()` with the restriction of treating the period `'.'` at the beginning of a string in a special manner. With this member function wild cards `'*'` and `'?'` do not match the period `'.'` at the beginning of a string. This member function assumes use in searches for file names. In addition, `pnmatch()` treats slash `'/'` and the period immediately following a slash in a special manner. This member function assumes use in searches for path names.

[11] The interval "{}", back reference "`\\n`", repetition expression "`+`" and character set "`\\w`" cannot be used.

PARAMETER

[I] `pat` Character pattern
[I] `pos` Position to start string matching
([I] : Input, [O] : Output)

RETURN VALUE

0	:	If a string inside an object matches `pat`.
Negative value (Error)	:	If a string inside an object does not match `pat`
	:	If `pos` has a value larger than the length of a string inside an object specified to it (Member functions 2, 4, 6, 8, 10, and 12).
	:	If there is no string inside an object.
	:	If `pat` is `NULL`.

EXAMPLE

The following code inquires whether a string that the object `my_str` includes matches the character pattern **.txt*, and then outputs the result to standard output:

```
stdstreamio sio;
tstring     my_str = "./Akari20090309.txt";

if (my_str.strmatch("*.txt") != 0) {
    sio.printf("The character string is not shown by \"*.txt\".\n");
}
else {
    sio.printf("The character string is shown by \"*.txt\".\n");
}
```

Result of execution

```
The character string is shown by "*.txt".
```

9.5.59 regmatch()

NAME

regmatch() — Attempts string matching using extended regular expression

SYNOPSIS

```
ssize_t regmatch( const char *pat, size_t *ret_span ) const;  ..... 1
ssize_t regmatch( size_t pos, const char *pat, size_t *ret_span ) const;  . 2
ssize_t regmatch( size_t pos, const char *pat, size_t *ret_span,
                  size_t *nextpos ) const;  ..... 3
ssize_t regmatch( const tstring &pat, size_t *ret_span ) const;  ..... 4
ssize_t regmatch( size_t pos, const tstring &pat, size_t *ret_span ) const;  5
ssize_t regmatch( size_t pos, const tstring &pat, size_t *ret_span,
                  size_t *nextpos ) const;  ..... 6
ssize_t regmatch( const tregex &pat, size_t *ret_span ) const;  ..... 7
ssize_t regmatch( size_t pos, const tregex &pat, size_t *ret_span ) const;  8
ssize_t regmatch( size_t pos, const tregex &pat, size_t *ret_span,
                  size_t *nextpos ) const;  ..... 9
```

DESCRIPTION

Attempts string matching of a string inside an object that uses a POSIX Extended Regular Expression (hereinafter referred to regular expression), and then returns the result.

Back reference information cannot be obtained with these member functions. If you wish to acquire the back reference information use of the regassign() member function for the tarray_tstring class (§10.4.13) is recommended.

Member functions 1 to 6 compile the regular expression `pat`, saves the result to an internal buffer that the functions encompass, and then perform the matching (If `pat` is the same as that previously compiled it is not recompiled again).

Member functions 7 to 9 specify the object for the tregex class retaining the result of compiling the regular expression. The regular expression therefore needs to be compiled in advance using the compile() member function for the tregex class before using the regmatch() member function (Refer to EXAMPLE-2).

In both cases if the function fails to compile the regular expression it outputs the content to standard error output.

String matching is attempted from position `pos` in a string inside an object to the right. The string matching is attempted within the range of up to where `'\0'` appears at the end of the string (When the newline character `'\n'` appears processing still does not terminate). If `pos` is not specified searches are made from the left end of a string inside an object. Please note that the lead position in strings is always 0.

The length of the matching string is returned to `*ret_span`. If you do not need information on the length of the string that matches `NULLL` can be used in `ret_span`.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. If `pat` matches a string and the length l of that matching string is 1 or more the position the same length as l to the right of the position of the matching string is returned as the variable referred to by `nextpos`. If the length of the matching string is 0 and the position one character to the right of the position of the matching string that is smaller than the length of the string itself that value is returned as the variable referred to by `nextpos`. In any other case than above the length of the string itself +1 is returned. If you do not need to acquire a value using `nextpos` NULL can also be used.

The basic unit of regular expressions is a regular expression that matches a single character. The methods of expressing strings for regular expression that are available for use with the character pattern `pat` are as shown in Table 21.

A character class can be specified inside lists. For the character classes that can be specified refer to the descriptions provided in §9.5.26. If you want the character `"]"` to be included in the targets for matching it must be positioned at the beginning of the list. The character `"^"` must also be positioned other than the beginning of a list, while the character `"-"` must be placed at the end of a list.

A back reference, when `"\\"` is followed by a decimal value character n that is not 0, matches the string that is the same as the sequence of characters matching the nth character in a parenthesized subexpression. The numbering for subexpressions is performed from the characters in which the position of an open parenthesis `"("` is to the left toward the characters in which the position of the parenthesis is to the right. For example, the string `"abc:def::abc:def"` matches the character pattern `"(\\w+):(\\w+)::\\1:\\2"`.

PARAMETER

Expression of a string for regular expression	Meaning
`"."`	Any single character other than a newline character
`"[...]"`	Single character contained in a list (referring to "[...]")
`"[^...]"`	Single character not contained in a list
`"a|b"`	Matches either "a" or "b"
`"(ab)"`	Matches "ab" group
`"\\w"`	Alphanumerical character (Equivalent to character class `[[:alnum:]]`)
`"\\W"`	Other than alphanumerical character (Equivalent to `[^[:alnum:]]`)
`"^"`	Beginning of a pattern line
`"$"`	End of pattern line
`"\\<"`	Null string at the beginning of a word
`"\\>"`	Null string at the end of a word
`"\\b"`	Null string beside a word
`"\\B"`	Null string other than beside a word
`"?"`	Repetition of previous character 0 times or once
`"*"`	Repetition of previous character 0 times or more
`"+"`	Repetition of previous character once or more
`"{n}"`	Repetition of previous character n times
`"{n,}"`	Repetition of previous character more than n times
`"{n,m}"`	Repetition of previous character more than n times but less than m times
`"\\n"`	Back reference

Table 21: List of methods of expressing strings for regular expressions.

[I]	`pos`	Position to start string matching
[I]	`pat`	Character pattern (regular expression) or compiled object for the tregex class
[O]	`ret_span`	Length of matching string
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative	:	Position of a string inside an object that matches `pat`
Negative value (Error)	:	If no string matches `pat`
	:	If there is not a string inside an object.
	:	If `pos` has a value larger than the length of a string inside an object specified to it.
	:	If `pat` is `NULL`.
	:	If the interval operators `{}` are not closed.
	:	If the list operators [] are not closed.
	:	If an unknown character class is set. ⟦For example use of *[:up:]*.⟧
	:	If a regular expression ends with a backslash.
	:	If the group operators () are not closed.
	:	If operators are used with an invalid range. ⟦For example use of *[9-0]*.⟧
	:	If an invalid back reference to the subexpression `\(...\)` is used.
	:	If an invalid back reference operator is used.
	:	If invalid use of pattern operators such as groups and lists are made. ⟦For example use of *[0-9)*.⟧
	:	If an invalid repetition operator is used such that `'*'` is the first character. ⟦For example use of *pat="*.txt"*.⟧

EXCEPTION

If the `regex` routine exhausted the memory.
If the system failed to secure an internal buffer.
If the system encountered any corrupt memory.

EXAMPLE-1

The following code searches a string that the object `my_str` includes for the position of a string with four consecutive numbers with the character pattern *[[:digit:]]{4}* in performing string matching. The result is then output to standard output:

```
stdstreamio sio;
tstring     my_str = "User ID : 1234";
size_t      t_span = 0;
ssize_t     t_ret = 0;

if ((t_ret = my_str.regmatch("[[:digit:]]{4}", &t_span)) < 0) {
    Error handling
}
else {
    sio.printf("%zd, %zu\n", t_ret, t_span);
}
```

Result of execution

```
10,4
```

EXAMPLE-2

The following code performs the same processing as in EXAMPLE-1 but with the difference that a regular expression is compiled in advance before `regmatch()` is used. Regular expressions are compiled using the compile() member function, as seen in `my_pat.complie("[[:digit:]]{4}")`. Any errors in the compilation processing are checked for using `cregex()`.

```
stdstreamio sio;
tregex      my_pat;
tstring     my_str = "User ID : 1234";
size_t      t_span = 0;
ssize_t     t_ret = 0;

my_pat.compile("[[:digit:]]{4}");   /* <-- The regular expression is compiled here */
if ( my_pat.cregex() == NULL ) {
    /* Error handling: Failed to compile the regular expression */
}
if ((t_ret = my_str.regmatch(my_pat, &t_span)) < 0) {
    Error handling
}
else {
    sio.printf("%zd, %zu\n", t_ret, t_span);
}
my_pat.init();                      /* Discard the result of compilation */
```

10 TARRAY_TSTRING class

The tarray_tstring class enables users to easily handle string arrays. Individual elements of an array can be the object for the tstring class (§9), and combined with tstring class APIs to provide string array APIs that are easy to use.

The class has the following characteristics:

- Memory is automatically secured, so that objects can be assigned immediately after being created.
- Pointer arrays (NULL-terminated) can be acquired to a string at any time, making it easy for functions in libc such as `execvp()` to be used.
- The notation for `printf()` can be used with many of the member functions.
- A wealth of the member functions for the tstring class (§9) are available for use through `[]` or at() member function.
- Easily divides space-delimited, TAB-delimited or CSV-format strings and arrays them.
- Member functions that enable users to edit all the elements of strings for arrays in one stroke are also available (e.g., `chomp()`, `trim()` etc). Those functions can be used in the same manner as with the tstring class (§9).
- - Provides the APIs for the search processing arrays that include POSIX Extended Regular Expressions.

If you wish to use the tarray_tstring class you must add "`#include <sli/tarray_tstring.h>`" to the code. In addition, if you also need to declare a namespace (§4.1) you must also add "`using namespace sli;`" to the code.

The following is a simple example of using the class.

```
#include <sli/stdstreamio.h>
#include <sli/asarray_tstring.h>
using namespace sli;

int main()
{
    stdstreamio sio;
    tarray_tstring my_arr;
    size_t i=0;
    my_arr[i++] = "MacOSX";                     /* Assign "MacOSX" to the element 0 */
    my_arr[i++].printf("Linux");                /* Assign "Linux" to the element 1 */
    my_arr.at(i).printf("Solaris");             /* Assign "Solaris" to the element 2 */
    for ( i=0 ; i < my_arr.length() ; i++ ) {   /* Display all the elements */
        sio.printf("%zu ... [%s]\n", i, my_arr.cstr(i));
    }
}
```

Result of execution

```
0 ... [MacOSX]
1 ... [Linux]
2 ... [Solaris]
```

10.1 Creating objects

There are the three methods of providing objects with an initial value[12].

The first method does not need any arguments to be specified.

```
tarray_tstring my_arr0;
```

With this neither a buffer for the string nor a buffer for the pointer array is secured.

The second method provides objects with a default value using a variable-length argument.

```
tarray_tstring my_arr0("tokyo", "osaka", "nagoya", NULL);
```

With this the array is initialized by the string it is given provided with. `NULL` must always be provided at the end of the argument.

The third method provides a NULL-terminated pointer array of the `char *[]` type. For example, the main() function can simply be provided with `argv`.

```
tarray_tstring my_arr0(argv);
```

With this the array inside the object is initialized by the string array `argv` that is provided.

10.2 List of member functions

Table 22 lists the member functions.

	Name of member function	Feature
§10.3.1	`[]`	Reference to a string object in a specified element
§10.3.2	`=`	Assigns string arrays
§10.3.3	`+=`	Addition of string arrays
§10.3.4	`+=`	Addition of string elements
§10.4.1	`length()`	Length of string array (number of strings) or length of value string
§10.4.2	`cstrarray()`	The address of a pointer array (NULL-terminated) for a value string
§10.4.3	`cstr()`, `c_str()`	Address for a value string in a specified element
§10.4.4	`at()`, `at_cs()`	Reference to a string object in a specified element
§10.4.5	`dprint()`	Outputs information on objects to standard error output (For debugging user programs)
§10.4.6	`copy()`	Copies (part of) arrays to an external object
§10.4.7	`swap()`	Interchange of objects
§10.4.8	`init()`	Complete initialization of objects
§10.4.9	`assign()`, `assignf()`	Initialization and assignment of objects (Specifies a single string)
§10.4.10	`assign()`, `vassign()`	Initialization and assignment of objects (Specifies multiple strings or a string array)
§10.4.11	`explode()`	Divides a string in an argument and assigns it to an array (simple & fast)
§10.4.12	`split()`	Divides a string in an argument and assigns it to an array (advanced edition)
§10.4.13	`regassign()`	Performs regular expression matching on strings in an argument, and assigns the result to an array
§10.4.14	`put()`, `putf()`	Sets n pieces of a string to any element position
§10.4.15	`put()`, `vput()`	Sets multiple strings or a string array to any element position
§10.4.16	`append()`, `appendf()`	Addition of elements (Specifies a single string)
§10.4.17	`append()`, `vappend()`	Addition of elements (Specifies multiple strings or a string array)

Table 22: List of member functions available for use with the tarray_tstring class (Continued on next page)

[12] The class does not include the operating modes the tstring class (§9) does.

	Name of member function	Feature
§10.4.18	`insert()`, `insertf()`	Insertion of elements (Specifies a single string)
§10.4.19	`insert()`, `vinsert()`	Insertion of elements (Specifies multiple strings or a string array)
§10.4.20	`replace()`, `replacef()`	Replacement of elements (Specifies a single string)
§10.4.21	`replace()`, `vreplace()`	Replacement of elements (Specifies multiple strings or a string array)
§10.4.22	`erase()`	Deletion of elements
§10.4.23	`clean()`	Pads all the element values for an existing array with any string
§10.4.24	`resize()`	Changes the length of an array
§10.4.25	`resizeby()`	Relatively changes the length of an array
§10.4.26	`crop()`	Cropping of arrays
§10.4.27	`chomp()`	Elimination of newline characters in all the elements
§10.4.28	`trim()`	Elimination of spaces at both ends of all the elements
§10.4.29	`ltrim()`	Elimination of a space at the left end of all the elements
§10.4.30	`rtrim()`	Elimination of a space at the right end of all the elements
§10.4.31	`strreplace()`	String search and replacement of all the elements
§10.4.32	`regreplace()`	String search and replacement of all the elements using a regular expression
§10.4.33	`tolower()`	Replaces the uppercase version of characters of all the elements with the lowercase version
§10.4.34	`toupper()`	Replaces the lowercase version of characters of all the elements with the uppercase version
§10.4.35	`expand_tabs()`	Replaces TAB characters in all the elements with a white space character
§10.4.36	`contract_spaces()`	Replaces white space characters in all the elements with a TAB character
§10.4.37	`find_elem()`	Searches from the left side (beginning) of an array element
§10.4.38	`rfind_elem()`	Searches from the right side (end) of an array element
§10.4.39	`find()`	Searches an array from the left side (beginning) for a string
§10.4.40	`rfind()`	Searches an array from the right side (end) for a string
§10.4.41	`find_matched_str()`	Searches for an element (string) that matches a pattern
§10.4.42	`find_matched_fn()`	Searches for an element (file name) that matches a pattern
§10.4.43	`find_matched_pn()`	Searches for an element (path name) that matches a pattern
§10.4.44	`regmatch()`[Normal edition]	Searches for a string using an extended regular expression
§10.4.45	`regmatch()`[Advanced edition]	Searches for a string using an extended regular expression

Table 22: List of member functions available for use with the tarray_tstring class (Continued from previous page)

10.3 Operators

10.3.1 []

NAME

`[]` — Reference to a string object in a specified element

SYNOPSIS

```
tstring &operator[]( size_t index );  ..... 1
const tstring &operator[]( size_t index ) const;  ..... 2
```

DESCRIPTION

Returns a reference to an array element (tstring class object (§9)) specified by an index. "`[]`" can be immediately followed by "`.`" to enable use of the tstring class member functions (§9) (In the EXAMPLE the "`=`" operator and the tstring class cstr() member function are used).

Member function 1 is for both reading and writing, and operates in the same manner as `at()` does. Member function 2 is for reading only, and operates in the same manner as `at_cs()` does.

If `index` has a value larger than the length of an array specified to it, with member function 1 the length of the array is automatically extended, but with member function 2 an exception occurs. Please note that the element number for the first element of arrays is always 0.

Whether member function 1 or 2 is used is automatically determined by the presence or absence of the "const" attribute of an object. Member function 1 is automatically selected if the object does not have a "const" attribute and member function 2 if it does.

For more details on `at()`, and `at_cs()` refer to §10.4.4.

PARAMETER

[I] `index` Element numbers starting from `0`

RETURN VALUE

Reference to the array element specified by an index

EXCEPTION

If the system failed to secure an internal buffer (Member function 1).
If `index` has a value larger than the length of an array specified to it (Member function 2).

EXAMPLE

The following code adds the string `"camellia"` to the string array object `my_arr` using the operators "`[]`", and then prints the result to standard output. Refer to the descriptions provided in §10.4.1 for more details on `length()`.

```
stdstreamio sio;

tarray_tstring my_arr("hawthorn", "oak", NULL);
my_arr[2] = "camellia";

/* Display */
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    sio.printf("[%s]\n", my_arr[i].cstr());
}
```

Result of execution

```
[hawthorn]
[oak]
[camellia]
```

10.3.2 =

NAME

= — Assigns string arrays

SYNOPSIS

```
tarray_tstring &operator=(const tarray_tstring &obj);  ..... 1
const char *const *operator=(const char *const *elements);  ..... 2
```

DESCRIPTION

Assignes object or string array specified on the right-hand side (argument) of the operator to itself.

PARAMETER

[I]	`obj`	tarray_tstring class object
[I]	`elements`	Address of pointer array for a string (`NULL`-terminated)

RETURN VALUE

Reference to itself (member function 1)
Pointer array to internal string buffer (member function 2)

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory (member function 1)

EXAMPLE

The following code assigns the content of the pointer array `menu` as a string array to the string array object `my_arr` using the operator "`=`", and then prints the result to standard output. Refer to the respective descriptions provided in §10.4.3 and §10.4.1 for more details on `cstr()` and `length()`.

```
stdstreamio sio;

const char *menu[] = {"rice ball", "sushi", "tofu", NULL};
tarray_tstring my_arr;
my_arr = menu;          /* '=' is the operator for the tarray_tstring class */

/* Display */
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    sio.printf("[%s]\n", my_arr.cstr(i));
}
```

Result of execution

```
[rice ball]
[sushi]
[tofu]
```

10.3.3 +=

NAME

+= — Addition of string arrays

SYNOPSIS

```
tarray_tstring &operator+=(const tarray_tstring &obj);  ..... 1
const char *const *operator+=(const char *const *elements);  ..... 2
```

DESCRIPTION

Adds to a string array the string array specified on the right (argument) of the operator.

PARAMETER

[I]	`obj`	tarray_tstring class object
[I]	`elements`	Address of pointer array for a string (`NULL`-terminated)

RETURN VALUE

Reference to itself (member function 1)
Pointer array to an internal string buffer (member function 2)

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory (member function 1).

EXAMPLE

The following code adds the string array `addTree` to the string array object `my_arr` using the operator "`+=`", and then prints the result to standard output. Refer to the respective descriptions provided in §10.4.3 and §10.4.1 for more details on `cstr()` and `length()`.

```
stdstreamio sio;

tarray_tstring my_arr("ginkgo", "Japanese apricot", "maple", NULL);

const char *addTree[] = {"oak", "cherry tree", NULL};
my_arr += addTree;

/* Display */
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    sio.printf("[%s]\n", my_arr.cstr(i));
}
```

Result of execution

```
[ginkgo]
[Japanese apricot]
[maple]
[oak]
[cherry tree]
```

10.3.4 +=

NAME

+= — Addition of string elements

SYNOPSIS

```
tarray_tstring &operator+=(const char *str);  ..... 1
tarray_tstring &operator+=(const tstring &str);  ..... 2
```

DESCRIPTION

Adds to a string array the string specified on the right (argument) of the operator.

PARAMETER

[I] `str` Address for string (Member function 1)
tstring class object (Member function 2)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code adds the string `"wisteria"` to the string array object `my_arr` using the operator "`+=`", and then prints the result to standard output. Refer to the respective descriptions provided in §10.4.3 and §10.4.1 for more details on `cstr()` and `length()`.

```
stdstreamio sio;

tarray_tstring my_arr("nandina", "elm", NULL);
my_arr += "wisteria";

/* Display */
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    sio.printf("[%s]\n", my_arr.cstr(i));
}
```

Result of execution

```
[nandina]
[elm]
[wisteria]
```

10.4 The member functions

General information

The size_t type handles numerical values as unsigned integer numbers. Providing a negative value to a member function that has the size_t type as the argument increases the likelihood of the program aborting. Ensure not to set a negative value.

10.4.1 length()

NAME

length() — Length of a string array (number of arrays) or the length of a value string

SYNOPSIS

```
size_t length() const;  ..... 1
size_t length( size_t index ) const;  ..... 2
```

DESCRIPTION

Returns the length of a string array (number of arrays) (member function 1).

If `index` is specified it returns the length of a string in the element corresponding to the element number in an argument (member function 2). Please note that the element number for the first element of arrays is always `0`.

RETURN VALUE

The number of string arrays (member function 1), or length of a string in the specified element (member function 2)

EXAMPLE

The following code prints to standard outputs the number of arrays in the string array object `my_arr`, and the length of the string in each of the elements. Refer to the description provided in §10.4.4 for more details on `at()`.

```
stdstreamio sio;

tarray_tstring my_arr;
my_arr.at(0).printf("Hello");
my_arr.at(1).printf("Hoge");

sio.printf("my_arr length = %zu\n",my_arr.length());
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr index%zu length = %zu\n",i, my_arr.length(i));
}
```

Result of execution

```
my_arr length = 2
my_arr index0 length = 5
my_arr index1 length = 4
```

10.4.2 cstrarray()

NAME

cstrarray() — Address of pointer array (NULL-terminated) for a value string

SYNOPSIS

```
const char *const *cstrarray() const;
```

DESCRIPTION

Returns the address of of the pointer array for a string for each element. Pointer arrays are always NULL-terminated.

RETURN VALUE

Pointer array (NULL-terminated) to a string buffer

EXAMPLE

The following code assigns `tmp` to the string array object `my_arr`, acquires the address of the pointer array for the string, and then prints the content of each of the elements to standard output:

```
stdstreamio sio;

/* Initialization of the object */
const char *tmp[] = {"linux", "windows", "mac", NULL};
tarray_tstring my_arr(tmp);

/* Acquire the address for the pointer array for the string */
const char *const *ptr = my_arr.cstrarray();
if ( ptr != NULL ) {
    int i;
    for ( i=0 ; ptr[i] != NULL ; i++ ) {
        sio.printf("%d ... [%s]\n", i, ptr[i]);
    }
}
```

Result of execution

```
0 ...  [linux]
1 ...  [windows]
2 ...  [mac]
```

An example of using the execvp function with the cstrarray() member function is provided in §3.4.3.

10.4.3 cstr(), c_str()

NAME

cstr(), c_str() — Address of value string in a specified element

SYNOPSIS

```
const char *cstr( size_t index ) const;  ..... 1
const char *c_str( size_t index ) const;  ..... 2
```

DESCRIPTION

Returns the beginning address of the element specified by `index` of a string array. If `index` has a value larger than the length of an array specified to it `NULL` is returned. Please note that the element number for the first element of arrays is always `0`.

`cstr()` and `c_str()` have different names but operate in the same manner.

RETURN VALUE

Beginning address of an element of a string array

EXAMPLE

The following code assigns `tmp` to the string array object `my_arr`, acquires the beginning address of each element of `my_arr`, and then prints the content to standard output:

```
stdstreamio sio;

/* Initialization of the object */
const char *tmp[] = {"linux", "windows", "mac", NULL};
tarray_tstring my_arr(tmp);

/* Display */
```

```
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    sio.printf("[%s]\n", my_arr.cstr(i));
}
```

Result of execution

```
[linux]
[windows]
[mac]
```

10.4.4 at(), at_cs()

NAME

at(), at_cs() — Reference to a string object in a specified element

SYNOPSIS

```
tstring &at( size_t index );  ..... 1
const tstring &at( size_t index ); const  ..... 2
const tstring &at_cs( size_t index ) const;  ..... 3
```

DESCRIPTION

Returns a reference to the array element (tstring class object (§9)) specified by `index`. These member functions can be immediately followed by "`.`" to enable use of the tstring class member functions (§9) (In the EXAMPLE the tstring class cstr() member function is used). Please note that the element number for the first element of arrays is always `0`.

Member function 1 is for both reading and writing strings, while member functions 2 and 3 are for reading only.

With the at() member function whether member function 1 or 2 is used is automatically determined by the presence or absence of the "const" attribute for an object. Member function 1 is automatically selected if the object does not have a "const" attribute while member function 2 is if it does.

With member function 1 `index` being larger than the length of the specified array results in a new array element being created and `""` (zero-length string) being assigned. No exceptions occur unless the system fails to secure a buffer.

With member functions 2 and 3 `index` being larger than the length of the specified array results in an exception occurring.

PARAMETER

[I] `index` Element number
([I] : Input, [O] : Output)

RETURN VALUE

Reference to the string (tstring) object corresponding to specified element number

EXCEPTION

If the system failed to secure an internal buffer (Member function 1)
If an `index` larger than the length of an array is specified (Member functions 2 and 3)

EXAMPLE

The following code assigns a string to the element with element number 0 of string array `my_arr`, and then prints the content of element numbers 0 and 1 of `my_arr` to standard output:

```
stdstreamio sio;

tarray_tstring my_arr;
my_arr.at(0) = "Hello";
sio.printf("my_arr[0] = %s\n",my_arr.at(0).cstr());
sio.printf("my_arr[1] = %s\n",my_arr.at(1).cstr());
```

Result of execution

```
my_arr[0] = Hello
my_arr[1] =
```

10.4.5 dprint()

NAME

dprint() — Outputs object information to standard error output (For user debugging)

SYNOPSIS

```
void dprint() const;
```

DESCRIPTION

Outputs object information to standard error output.

Member function designed for debugging user programs.

EXAMPLE

The following code outputs the information of the object `my_array` to standard error output. The result of execution results in the address for the object displayed in `[]`, and will depend on the environment:

```
tarray_tstring my_array("MZ-80B", "MZ-2861", "X1C", "X1 turboZ", NULL);
my_array.dprint();
```

Result of execution

```
sli::tarray_tstring[obj=0x7fbffff3e0] = {"MZ-80B", "MZ-2861", "X1C", "X1 turboZ"}
```

10.4.6 copy()

NAME

copy() — Copies (part of) string arrays to another string array

SYNOPSIS

```
ssize_t copy( tarray_tstring *dest ) const;  ................................ 1
ssize_t copy( size_t index, tarray_tstring *dest ) const;  ................. 2
ssize_t copy( size_t index, size_t n, tarray_tstring *dest ) const;  ..... 3
```

DESCRIPTION

Copies all or part of string arrays to the other string array object `dest`. The return value is the number of elements to be written to `dest`.

Member functions 1 copies all string arrays to `dest`.

Member functions 2 and 3 copy elements starting from the element number `index` of string arrays. Please note that the element number for the first element of arrays is always `0`. In addition, member functions 3 enables you to specify the number `n` of elements to be copied.

If the value of `index` + `n` is larger than the number of elements to copy only the elements from the index position through to the last element are then copied. If the value of `index` is larger than the number of elements to copy the content of `dest` is erased, and the return value will be `-1`.

PARAMETER

[O]	`dest`	tarray_tstring class object to copy to
[I]	`index`	Position of an array element in an object to copy from
[I]	`n`	Number of elements to be copied

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	Number of elements copied
Negative value (Error)	:	If `index` has a value larger than the length of a string array specified to it.

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code copies two elements starting from element number 2 of the string array `my_menu` to the string array `dest_arr`, and then prints the content of `dest_arr` to standard output:

```
stdstreamio sio;

tarray_tstring my_menu("pickles", "natto", "tempura", "sukiyaki", NULL);
tarray_tstring dest_arr;

my_menu.copy(2, 2, &dest_arr);
for ( size_t i = 0 ; i < dest_arr.length() ; i++ ) {
    sio.printf("dest_arr[%zu] = %s\n", i, dest_arr.cstr(i));
}
```

Result of execution

```
dest_arr[0] = tempura
dest_arr[1] = sukiyaki
```

10.4.7 swap()

NAME

swap() — Interchanging of string array objects

SYNOPSIS

```
tarray_tstring &swap( tarray_tstring &sobj );
```

DESCRIPTION

Interchanges the content of the string array object `sobj` with the content of its own.

PARAMETER

[I/O]	`sobj`	tarray_tstring class object to be interchanged

([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXAMPLE
The following code interchanges the string array `myMenu_arr` with the string array `myTree_arr`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring myMenu_arr("rice ball", "sushi", "tofu", NULL);
tarray_tstring myTree_arr("Pine", "Ginkgo", "Magnolia", NULL);

myMenu_arr.swap(myTree_arr);
for ( size_t i = 0 ; i < myMenu_arr.length() ; i++ ) {
    sio.printf("myMenu_arr[%zu] = %s\n", i, myMenu_arr.cstr(i));
}
```

Result of execution
```
myMenu_arr[0] = Pine
myMenu_arr[1] = Ginkgo
myMenu_arr[2] = Magnolia
```

10.4.8 init()

NAME
init() — Complete initialization of objects

SYNOPSIS
`tarray_tstring &init();` 1
`tarray_tstring &init(const tarray_tstring &obj);` 2

DESCRIPTION
Initializes string arrays.

Member function 1 completely initializes string arrays. The memory area allocated to the array and string buffer of a string array object then gets entirely released, and hence execution of the `cstrarray()` member function (§10.4.2) returns `NULL`.

Member function 2 initializes objects with the content of `obj` (copies all the content of `obj` to itself).

PARAMETER
[I] `obj` tarray_tstring class object
([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXCEPTION
If the system failed to secure an internal buffer.
If the system encountered any corrupt memory (Member function 2).

EXAMPLE

The following code initializes the string array object `my_arr` with `my_treeArr`, and then prints the result to standard output. In addition, it completely initializes it with `init()`, and then prints the array length to standard output:

```
stdstreamio sio;

tarray_tstring my_treeArr("pine", "willow", NULL);
tarray_tstring my_arr;

my_arr.init(my_treeArr);
/* Display */
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("[%s]\n", my_arr.cstr(i));
}

/* Completely initialize */
my_arr.init();
/* Display */
sio.printf("my_arr.length = [%zu]\n",  my_arr.length());
```

Result of execution

```
[pine]
[willow]
my_arr.length = [0]
```

10.4.9 assign(), assignf(), vassignf()

NAME

assign(), assignf(), vassignf() — Initialization and assignment of objects (Specifies a single string)

SYNOPSIS

```
tarray_tstring &assign( const char *str, size_t n );  ..................... 1
tarray_tstring &assign( const tstring &str, size_t n );  .................. 2
tarray_tstring &assignf( size_t n, const char *format, ... );  ............ 3
tarray_tstring &vassignf( size_t n, const char *format, va_list ap );  .... 4
```

DESCRIPTION

Sets the number of array elements in an object using `n`, and assigns to all the elements a specified string.

Member functions 1 and 2 assign the string `str` to n array elements.

Member functions 3 and 4 assign to n array elements strings created according to the format specified by `format`. Member function 3 converts each element of data of a variable-length argument using the conversion specifications in `format`. Member function 4 converts the list `ap` of variable-length arguments using the conversion specifications in `format`. For more information on `format` refer to the descriptions provided in §8.1.14.

PARAMETER

[I]	`n`	Number of elements of an array
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of a variable-length argument that supports `format`
[I]	`ap`	List of variable-length arguments that support `format`

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted into the specified conversion format (Member functions 3 and 4).

EXAMPLE

The following code initialize objects with the string "`***`" + "`Japanese quince`"using the format specifications, and creates string array `my_arr` with three elements. It then prints the result to standard output for use in verification:

```
stdstreamio sio;

tarray_tstring my_arr;
my_arr.assignf(3, "***%s", "Japanese quince");
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = ***Japanese quince
my_arr[1] = ***Japanese quince
my_arr[2] = ***Japanese quince
```

10.4.10 assign(), vassign()

NAME

assign(), vassign() — Initialization and assignment of objects (Specifies multiple strings or a string array)

SYNOPSIS

```
tarray_tstring &assign( const char *el0, const char *el1, ... );  ........ 1
tarray_tstring &vassign( const char *el0, const char *el1, va_list ap );   2
tarray_tstring &assign( const char *const *elements );  ..................... 3
tarray_tstring &assign( const char *const *elements, size_t n );  ........ 4
tarray_tstring &assign( const tarray_tstring &src, size_t idx2 = 0 );  .... 5
tarray_tstring &assign( const tarray_tstring &src, size_t idx2,
                        size_t n2 );  ....................................... 6
```

DESCRIPTION

Initializes string arrays inside an object with the multiple strings specified by `el0, el1, ...` or the string arrays specified by `elements` and `src`.

Member functions 1 and 2 specify `el0`, `el1` and the variable-length argument or the list `ap` of variable-length arguments. Variable-length arguments must be `NULL`-terminated.

Member functions 3 and 4 specify to the argument `elements` the pointer array for a string. Pointer arrays must be `NULL`-terminated with member function 3. With member function 4 the number of elements can be specified by `n`. If an `n` larger than the number of `elements` (until reaching `NULL`) is specified `n` is ignored.

Member functions 5 and 6 enable `idx2` to be used to specify the element position to start the string array `src` to be sourced, with the number of elements being provided by `n2`. Member function 5 can be used without specifying `idx2`. However, the function will be processed as though `0` had been specified. Member function 6 enables specification of the number `n2` of elements to be sourced. Please note that the element number for the first element of arrays is always `0`.

PARAMETER

[I]	`el0`	String that is placed in an element (0th)
[I]	`el1`	String that is placed in an element (The first)
[I]	`...`	String that is placed in an element (The second and any following need to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for a string that is placed in an element (The second and any following need to be `NULL`-terminated)
[I]	`elements`	Pointer array for a string that is placed in an element (With member function 3 `NULL`-terminated)
[I]	`n`	Number of elements of the array `elements`
[I]	`src`	tarray_tstring class object that includes the string array to be sourced
[I]	`idx2`	Position to start an element in `src` (When assigning a sub-array in `src`)
[I]	`n2`	Number of elements in `src` (When assigning a sub-array in `src`)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXAMPLE

The following code initializes the string array `my_arr` using two elements starting from the element number 1 of the string array `myTree`. It then prints the result to standard output for use in verification:

```
stdstreamio sio;

const tarray_tstring myTree("fir", "magnolia", "dogwood", NULL);
tarray_tstring my_arr;

/*  Two elements starting from the element number 1 of the array myTree */
my_arr.assign(myTree,1,2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("*** my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
*** my_arr[0] = magnolia
*** my_arr[1] = dogwood
```

10.4.11 explode()

NAME

explode() — Divides a string and assigns it to a string array (simple edition)

SYNOPSIS

```
tarray_tstring &explode( const char *src_str, const char *delim );  ..... 1
tarray_tstring &explode( const tstring &src_str, const char *delim )  ..... 2
```

DESCRIPTION

Divides the string `src_str` with the delimiter string `delim`, and then assigns it to a string array. The delimiter string should be placed between two elements, therefore, elements of zero-length string can exist in the result.

Compared with split() member function (§10.4.12), explode() works faster than it.

PARAMETER

[I] `src_str` String to be divided
[I] `delim` Delimiter string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

See EXAMPLE of split() member function. (§10.4.12)

10.4.12 split()

NAME

split() — Divides a string and assigns it to a string array (advanced edition)

SYNOPSIS

```
tarray_tstring &split( const char *src_str, const char *delims,
                       bool zero_str, const char *quotations,
                       int escape, bool rm_escape );  ..... 1
tarray_tstring &split( const tstring &src_str, const char *delims,
                       bool zero_str, const char *quotations,
                       int escape, bool rm_escape );  ..... 2
tarray_tstring &split( const char *src_str, const char *delims,
                       bool zero_str = false );  ..... 3
tarray_tstring &split( const tstring &src_str, const char *delims,
                       bool zero_str = false );  ..... 4
```

DESCRIPTION

Divides the string `src_str` with the delimiter characters and then assigns it to a string array. The delimiter characters are given by character set of `delims` argument, and `delims` can be specified as a simple list of characters like "`" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

`zero_str` can be specified to indicate whether a string length of zero is allowable as an element after division. If `zero_str` is `false` elements with a string length of 0 cannot be created. If `zero_str` is `true` elements with a string length of `0` can created (used for the csv format, etc.). If `zero_str` is not specified it is treated as `false`.

If you do not want to divide strings that are parenthesized with "specific characters" such as quotation marks etc., such "specific characters" can be specified using `quotations`. For example, if you wanted to exclude any strings parenthesized with a single quotation from the strings to be divided `"'"` would be used.

An escape character can be specified using `escape`. If you want to delete any escape characters remaining after the division set `rm_escape` to `true`. However, any escape characters in the strings that are parenthesized by a character specified using `quotations` cannot be deleted.

PARAMETER

[I]	`src_str`	String to be divided
[I]	`delims`	String that includes delimiter characters
[I]	`zero_str`	Whether or not to allow string elements with a length of `0` after division (true/false)
[I]	`quotations`	String that includes quotation characters
[I]	`escape`	Escape character
[I]	`rm_escape`	Flag used to indicate whether or not to delete escape characters (true/false)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code divides the string `line` with the string `" "`, assigns the result to the object `my_arr`, and then prints the result to standard output:

```
stdstreamio sio;

const char *line = "Fragrant olive is 'KINMOKUSEI'. It is good smelling.";
tarray_tstring my_arr;
my_arr.split(line, " ", false);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] ===> %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] ===> Fragrant
my_arr[1] ===> olive
my_arr[2] ===> is
my_arr[3] ===> 'KINMOKUSEI'.
my_arr[4] ===> It
my_arr[5] ===> is
my_arr[6] ===> good
my_arr[7] ===> smelling.
```

An example of using member function 1 is provided in §3.4.7.

An example of dividing CSV-format strings is described in the EXAMPLE provided in §10.4.28.

10.4.13 regassign()

NAME

regassign() — Performs regular expression matching on strings in an argument, and then assigns the result to an array

SYNOPSIS

```
tarray_tstring &regassign( const char *src_str, const char *pat );  ..... 1
tarray_tstring &regassign( const char *src_str, size_t pos,
                           const char *pat );  ............................. 2
tarray_tstring &regassign( const char *src_str, size_t pos,
                           const char *pat, size_t *nextpos );  ........... 3
tarray_tstring &regassign( const tstring &src_str, const char *pat );  .... 4
tarray_tstring &regassign( const tstring &src_str, size_t pos,
                           const char *pat );  ............................. 5
tarray_tstring &regassign( const tstring &src_str, size_t pos,
                           const char *pat, size_t *nextpos );  ........... 6
tarray_tstring &regassign( const char *src_str, const tstring &pat );  .... 7
tarray_tstring &regassign( const char *src_str, size_t pos,
                           const tstring &pat );  ......................... 8
tarray_tstring &regassign( const char *src_str, size_t pos,
                           const tstring &pat, size_t *nextpos );  ........ 9
tarray_tstring &regassign( const tstring &src_str, const tstring &pat);   10
tarray_tstring &regassign( const tstring &src_str, size_t pos,
                           const tstring &pat );  ........................ 11
tarray_tstring &regassign( const tstring &src_str, size_t pos,
                           const tstring &pat, size_t *nextpos );  ....... 12
tarray_tstring &regassign( const char *src_str, const tregex &pat );  .... 13
tarray_tstring &regassign( const char *src_str, size_t pos,
                           const tregex &pat );  ......................... 14
tarray_tstring &regassign( const char *src_str, size_t pos,
                           const tregex &pat, size_t *nextpos );  ........ 15
tarray_tstring &regassign( const tstring &src_str, const tregex &pat );   16
tarray_tstring &regassign( const tstring &src_str, size_t pos,
                           const tregex &pat );  ......................... 17
tarray_tstring &regassign( const tstring &src_str, size_t pos,
                           const tregex &pat, size_t *nextpos );  ........ 18
```

DESCRIPTION

Attempts string matching on the string `src_str` that uses a POSIX Extended Regular Expression (hereinafter referred to as regular expression) specified by `pat`, and if a string matches the expression stores a substring that can be back-referenced to an array inside the object (length of the array is 1 or more). If no string matches the expression or the processing encountered an error due to the reason that the regular expression was incorrect etc. (for more details refer to the RETURN VALUE item in §9.5.59) the array is initialized, and no string assigned to the array (length of the array is 0).

The position of the matching string can be acquired using the `reg_pos()` member function. The prototype for that is as follows:

```
size_t reg_pos( size_t idx ) const;
```

The element numbers starting from 0 are specified using the `idx` argument. The element with element number 0 is used to store the information on the entire matching string, while the elements with the element number 1 and later are used to individually store the information of the substring that matches the regular expressions `(...)` (i.e. back reference information).

Member functions 1 to 12 compile the regular expression `pat`, save the result to an internal buffer that the functions encompass, and then perform the matching (When `pat` is the same as previously compiled it is not recompiled again).

Member functions 13 and 18 specify an object for the tregex class to hold the result of compiling the regular expression. The regular expression therefore needs to be compiled in advance using the compile() member function of the tregex class before use of the regassign() member function (Refer to EXAMPLE).

In both cases if the function fails to compile the regular expression it outputs the content to standard error output.

If `pos` is not specified matching is attempted from the left end of the string `src_str`, while if `pos` is specified matching is attempted from the position `pos` in the string `src_str`. The string matching is attempted within the range of up to where `'\0'` appears at the end of the string (If the newline character `'\n'` appears the processing still does not terminate). Please note that the lead position in strings is always 0.

If you want to continuously search for characters or strings `nextpos` can be used to acquire the value that should be provided to `pos` in the next iteration. If a string matches the expression the position the same length as the matching string (the same length as the character if the length of the matching string is 0) to the right of that position is returned as the variable referred to by `nextpos`, and if no string matches the expression the length of the string `src_str` +1 is returned. If you do not need `nextpos` NULL can also be used.

For more details on regular expressions refer to §9.5.59.

PARAMETER

[I]	`src_str`	String on which to perform matching
[I]	`pos`	Position to start string matching
[I]	`pat`	Character pattern (regular expression) or compiled object for the tregex class
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the `regex` routine exhausted the memory.
If the system failed to secure an internal buffer.
If the system encountered any corrupt memory.

EXAMPLE

The following code retrieves the keyword and value for the string `"OS = linux"`. The part of the string that matches all of `my_pat is placed in my_elms.cstr(0)`, and the back-referenced substrings are stored in `my_elms.cstr(1)` and thereafter. Regular expressions are compiled using the compile() member function, as seen in

`my_pat.complie("([^ ]+)([ ]*=[ ]*)([^ ]+)")`. Any errors in the compilation process are identified using `cregex()`.

```
stdstreamio sio;
tarray_tstring my_elms;
tstring my_str = "OS = linux";
tregex my_pat;

my_pat.compile("([^ ]+)([ ]*=[ ]*)([^ ]+)");
if ( my_pat.cregex() == NULL ) {
    /* Error handling: Failed to compile the regular expression */
}
my_elms.regassign(my_str, my_pat);
if ( my_elms.length() == 4 ) {
    sio.printf("keyword=[%s] value=[%s]\n",
                my_elms.cstr(1), my_elms.cstr(3));
}
```

Result of execution

```
keyword=[OS] value=[linux]
```

An example of using member function 4 is provided in §3.4.8.

10.4.14 put(), putf(), vputf()

NAME

put(), putf(), vputf() — Sets n pieces of a string to any element position

SYNOPSIS

```
tarray_tstring &put( size_t index, const char *str, size_t n );  ..... 1
tarray_tstring &put( size_t index, const tstring &str, size_t n );  ..... 2
tarray_tstring &putf( size_t index, size_t n, const char *format, ... );   3
tarray_tstring &vputf( size_t index, size_t n, const char *format,
                    va_list ap );  ..................................... 4
```

DESCRIPTION

Writes `n` pieces of a specified string to the element number `index` position in a string array. Please note that the element number for the first element of arrays is always `0`.

`index` can take any value. If the number of elements in the array is smaller than specified by the argument the size of the array is automatically increased. An array having no elements results in `my_arr.put(0,"",6)` and `my_arr.put(2,"",4)`, for example, being the same. If an array has 4 elements and `my_arr.put(2,"",4)` is used on it then the number of elements will be 6, and elements number 2 and later `""`.

Member functions 1 and 2 write the string `str` to n elements starting from the element number provided by `index` of an array. If the length of the array before writing is smaller than `index + n` the length of the array after writing is increased to `index + n`.

Member function 3 writes each element of data of a variable-length argument, while member function 4 writes the strings created by converting the list `ap` of variable-length arguments depending on the conversion specifications set in `format` respectively to n elements starting from the element number provided by `index`. For more details on `format` refer to the descriptions provided in §8.1.14.

PARAMETER

[I]	`index`	Position to write to in an array inside an object
[I]	`n`	Number of elements
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for a string to be sourced
[I]	`...`	Each element of data of a variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted into the conversion format specified (Member functions 3 and 4).

EXAMPLE

The following code writes the string `"elm"` to two elements starting from the element number 1 of the string array `my_arr`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring my_arr;

my_arr.put(1, "elm", 2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] =
my_arr[1] = elm
my_arr[2] = elm
```

10.4.15 put(), vput()

NAME

put(), vput() — Sets multiple strings or a string array to any element position

SYNOPSIS

```
tarray_tstring &put( size_t index, const char *el0,
                     const char *el1, ... );  ..............................  1
tarray_tstring &vput( size_t index, const char *el0, const char *el1,
                      va_list ap );  ........................................  2
tarray_tstring &put( size_t index, const char *const *elements );  ........  3
tarray_tstring &put( size_t index, const char *const *elements,
                     size_t n );  ..........................................  4
tarray_tstring &put( size_t index, const tarray_tstring &src,
                     size_t idx2 = 0 );  ...................................  5
tarray_tstring &put( size_t index, const tarray_tstring &src,
                     size_t idx2, size_t n2 );  ............................  6
```

DESCRIPTION

Writes (overwrites) the multiple strings specified by `el0, el1, ...` or the string array specified by `elements` and `src` to element position `index` and then later in a string array.

`index` can take any value. If the number of elements in the array is smaller than specified by the argument the size of the array is automatically increased.

Member functions 1 and 2 specify `el0`, `el1` and the variable-length argument or list `ap` of variable-length arguments. Variable-length arguments must be `NULL`-terminated.

Member functions 3 and 4 specify to the argument `elements` a pointer array for a string. With member function 3 pointer arrays must be `NULL`-terminated. With member function 4 the number of elements can be specified by `n`. If an `n` value larger than the number of `elements` (until reaching `NULL`) is specified `n` is ignored.

Member functions 5 and 6 enable `idx2` to be used to specify the element position to start the string array `src` to be sourced, and the number of elements by `n2`. Member function 5 can be used without specifying `idx2`. However, the function is processed as though `0` had been specified. Member function 6 enables the number `n2` of the elements to be sourced to be specified. Please note that the element number for the first element of arrays is always `0`.

PARAMETER

[I]	`index`	Position to write to in an array inside an object
[I]	`el0`	String to be sourced (0th)
[I]	`el1`	String to be sourced (first)
[I]	`...`	String to be sourced (The second and following need to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for a string to be sourced (The second and following need to be `NULL`-terminated)
[I]	`elements`	Pointer array for a string to be sourced (With member function 3 must be `NULL`-terminated)
[I]	`n`	Number of elements of the array `elements`
[I]	`src`	tarray_tstring class object that includes the string array to be sourced
[I]	`idx2`	Position to start an element in `src` (When assigning a sub-array in `src`)
[I]	`n2`	Number of elements in `src` (When assigning a sub-array in `src`)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer

EXAMPLE

The following code writes two elements starting from element number 1 of the string array `tree_arr` to elements with the elements number 2 and later of the string array `my_arr`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring my_arr;

const char *mytree[] = {"maple", "larch", "camphor", NULL};
const tarray_tstring tree_arr(mytree);
```

```
my_arr.put(2, tree_arr, 1, 2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] =
my_arr[1] =
my_arr[2] = larch
my_arr[3] = camphor
```

10.4.16 append(), appendf(), vappendf()

NAME

append(), appendf(), vappendf() — Addition of elements (Specifies a single string)

SYNOPSIS

```
tarray_tstring &append( const char *str, size_t n );  ..... 1
tarray_tstring &append( const tstring &str, size_t n );  ..... 2
tarray_tstring &appendf( size_t n, const char *format, ... );  ..... 3
tarray_tstring &vappendf( size_t n, const char *format, va_list ap );  .... 4
```

DESCRIPTION

Adds `n` pieces of a specified string to the end of a string array. Please note that the element number for the first element of arrays is always `0`.

Member functions 1 and 2 add n pieces of the string `str` to a string array.

Member function 3 and 4 add to a string array n pieces of the string created according to the format specified in `format`. Member function 3 converts each element of data of a variable-length argument, while member function 4 converts the list `ap` of variable-length arguments depending on the format specification. For more details on `format` refer to the description in §8.1.14.

PARAMETER

[I]	`n`	Number of elements to be added
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for a string to be sourced
[I]	`...`	Each element of data of a variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted into the conversion format specified (Member functions 3 and 4).

EXAMPLE

The following code adds a string to the string array `my_arr`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring my_arr("maple", "larch", NULL);

const tstring mytrr = "gardenia";
my_arr.append(mytrr,2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = maple
my_arr[1] = larch
my_arr[2] = gardenia
my_arr[3] = gardenia
```

10.4.17 append(), vappend()

NAME

append(), vappend() — Addition of elements (Specifies multiple strings or a string array)

SYNOPSIS

```
tarray_tstring &append( const char *el0, const char *el1, ... );  ........ 1
tarray_tstring &vappend( const char *el0, const char *el1,
                         va_list ap );  ..................................... 2
tarray_tstring &append( const char *const *elements );  ................... 3
tarray_tstring &append( const char *const *elements, size_t n );  ........ 4
tarray_tstring &append( const tarray_tstring &src, size_t idx2 = 0 );  .... 5
tarray_tstring &append( const tarray_tstring &src, size_t idx2,
                        size_t n2 );  ..................................... 6
```

DESCRIPTION

Adds the multiple strings specified by `el0, el1, ...` or the string array specified by `elements` and `src` to the end of the array and later.

Member functions 1 and 2 specify `el0`, `el1` and the variable-length argument or the list `ap` of variable-length arguments. Variable-length arguments must be `NULL`-terminated.

Member functions 3 and 4 specify to the argument `elements` a pointer array for a string. ith member function 3 pointer arrays must be `NULL`-terminated. With member function 4 the number of elements can be specified by `n`. If `n` larger than the number of `elements` (until reaching `NULL`) specified then `n` is ignored.

Member function 5 and 6 enable `idx2` to be used to specify the element position to start the string array `src` to be sourced, and the number of elements by `n2`. Member function 5 can be used without specifying `idx2`. However, the function is processed as though `0` had been specified. Member function 6 enables the number `n2` of elements to be sourced to be specified. Please note that the element number for the first element of arrays is always `0`.

PARAMETER

[I]	el0	String to be sourced (0th)
[I]	el1	String to be sourced (first)
[I]	...	String to be sourced (The second and following need to be NULL-terminated)
[I]	ap	List of variable-length arguments for a string to be sourced (The second and following need to be NULL-terminated)
[I]	elements	Pointer array for a string to be sourced (With member function 3 must be NULL-terminated)
[I]	n	Number of elements of the array
[I]	src	tarray_tstring class object that includes the string array to be sourced
[I]	idx2	Position to start an element in src (When assigning a sub-array in src)
[I]	n2	Number of elements in src (When assigning a sub-array in src)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code adds to the string array my_arr two elements starting from the element number 2 of the string array tree_arr, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring tree_arr("chestnut", "zelkova", "crape myrtle", "daphne", NULL);
tarray_tstring my_arr;
my_arr.at(0) = "chestnut";

my_arr.append(tree_arr,2,2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = chestnut
my_arr[1] = crape myrtle
my_arr[2] = daphne
```

10.4.18 insert(), insertf(), vinsertf()

NAME

insert(), insertf(), vinsertf() — Insertion of elements (Specifies a single string)

SYNOPSIS

```
tarray_tstring &insert( size_t index, const char *str, size_t n );  ..... 1
tarray_tstring &insert( size_t index, const tstring &str, size_t n );  .... 2
tarray_tstring &insertf( size_t index, size_t n, const char *format, ... );  3
tarray_tstring &vinsertf( size_t index, size_t n,
                          const char *format, va_list ap );  ............... 4
```

DESCRIPTION

Inserts `n` pieces of a specified string to the position with the element number `index`in a string array. Please note that the element number for the first element of arrays is always `0`.

If a value larger than the length of an array is specified to `index` the assumption is made that the length of the array for the function itself is provided to `index`.

Member functions 1 and 2 insert n pieces of the string `str` to a string array.

Member functions 3 and 4 insert into a string array n pieces of a string created according to the format specified by `format`. Member function 3 converts each element of data of a variable-length argument, while member function 4 converts the list `ap` of variable-length arguments depending on the format specification. For more details on `format` refer to the description in §8.1.14.

PARAMETER

[I]	`index`	Position to insert to in an array inside an object
[I]	`n`	Number of elements to be added
[I]	`str`	String to be sourced
[I]	`format`	Format specifications for a string to be sourced
[I]	`...`	Each element of data of a variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXAMPLE

The following code inserts the strings `"palm"` and `"fir"` to the element position with the element number 1 in the string array `my_arr`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring my_arr("cycad", "dogwood", NULL);

my_arr.insert(1, "palm", "fir", NULL);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = cycad
my_arr[1] = palm
my_arr[2] = fir
my_arr[3] = dogwood
```

10.4.19 insert(), vinsert()

NAME

insert(), vinsert() — Insertion of elements (Specifies multiple strings or a string array)

SYNOPSIS

```
tarray_tstring &insert( size_t index, const char *el0,
                        const char *el1, ... );  ............................ 1
```

```
tarray_tstring &vinsert( size_t index, const char *el0,
                         const char *el1, va_list ap );  ..................  2
tarray_tstring &insert( size_t index, const char *const *elements );  .....  3
tarray_tstring &insert( size_t index, const char *const *elements,
                        size_t n );  ........................................  4
tarray_tstring &insert( size_t index, const tarray_tstring &src,
                        size_t idx2 = 0 );  .................................  5
tarray_tstring &insert( size_t index, const tarray_tstring &src,
                        size_t idx2, size_t n2 );  ..........................  6
```

DESCRIPTION

Inserts the multiple strings specified by `el0, el1, ...` or the string array specified by `elements` and `src` to the specified position `index` of a string array inside an object.

If a value larger than the length of the array for the function itself is specified to `index` the assumption is made that the length of the array for the function itself is provided to `index`.

Member functions 1 and 2 specify `el0`, `el1` and the variable-length argument or the list `ap` of variable-length arguments. Variable-length arguments must be `NULL`-terminated.

Member functions 3 and 4 specify to the argument `elements` a pointer array for a string. With member function 3 pointer arrays must be `NULL`-terminated. With member function 4 the number of elements can be specified by `n`. If `n` larger than the number of `elements` (until reaching `NULL`) specified then `n` is ignored.

Member functions 5 and 6 enable `idx2` to be used to specify the element position to start the string array `src` to be sourced, and the number of elements by `n2`. Member function 5 can be used without specifying `idx2`. However, the function is processed as though `0` had been specified. Member function 6 enables the number `n2` of elements to be sourced to be specified. Please note that the element number for the first element of arrays is always `0`.

PARAMETER

[I]	`index`	Position to insert to in an array inside an object
[I]	`el0`	String to be sourced (0th)
[I]	`el1`	String to be sourced (first)
[I]	`...`	String to be sourced (The second and following need to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for a string to be sourced (The second and following need to be `NULL`-terminated)
[I]	`elements`	Pointer array for a string to be sourced (With member function 3 must be `NULL`-terminated)
[I]	`n`	Number of `elements` of the array
[I]	`src`	tarray_tstring class object that includes the string array to be sourced
[I]	`idx2`	Position to start an element in `src` (When assigning a sub-array in `src`)
[I]	`n2`	Number of elements in `src` (When assigning a sub-array in `src`)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code inserts to the element position with the element number 1 in the string array `my_arr` two elements starting from the element number 0 of `addTree`, and then prints the result to standard output:

```
stdstreamio sio;

const char *addTree[] = {"cycad", "dogwood", NULL};
tarray_tstring my_arr("hawthorn", "oak", NULL);

my_arr.insert(1, addTree, 2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = hawthorn
my_arr[1] = cycad
my_arr[2] = dogwood
my_arr[3] = oak
```

10.4.20 replace(), replacef(), vreplacef()

NAME

replace(), replacef(), vreplacef() — Replacement of elements (Specifies a single string)

SYNOPSIS

```
tarray_tstring &replace( size_t idx1, size_t n1,
                         const char *str, size_t n2 );  ..................  1
tarray_tstring &replace( size_t idx1, size_t n1,
                         const tstring &str, size_t n2 );  ................  2
tarray_tstring &replacef( size_t idx1, size_t n1,
                          size_t n2, const char *format, ... );  ..........  3
tarray_tstring &vreplacef( size_t idx1, size_t n1,
                           size_t n2, const char *format, va_list ap );  .  4
```

DESCRIPTION

Replaces `n1` elements starting from the element position `idx1` in a string array with `n2` pieces of a specified string. Please note that the element number for the first element of arrays is always `0`.

If `idx1` has a value larger than the number of array elements specified to it the function performs the same processing as the `append()` member function (§10.4.16). If the sum of `idx1` and `n1` is larger than the number of elements of the array or the array needs to be expanded or contracted because of the size comparison between `n1` and `n2` the number of the elements is automatically adjusted.

Member functions 1 and 2 replace `n1` elements starting from the element number `idx1` in a string array with `n2` pieces of the string `str`.

Member functions 3 and 4 replace `n1` elements starting from the element number `idx1` of a string array with `n2` pieces of a string created according to the format specified by `format`.

Member function 3 converts each element of data of a variable-length argument, while member function 4 converts the list `ap` of variable-length arguments depending on the format specifications. For more details on `format` refer to the descriptions provided in §8.1.14.

PARAMETER

[I]	`idx1`	Position to start an array inside an object
[I]	`n1`	Number of elements to be replaced
[I]	`n2`	Number of elements to which a specified string is assigned
[I]	`format`	Format specifications for string to be sourced
[I]	`...`	Each element of data of a variable-length argument supporting `format`
[I]	`ap`	List of variable-length arguments supporting `format`
[I]	`str`	String to be sourced

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If each element of data of a variable-length argument is a value that cannot be converted into the conversion format specified (Member functions 3 and 4).

EXAMPLE

The following code replaces 1 element starting from the element number 1 of the string array `my_arr` with the string `"linden"`, and then prints the result to standard output:

```
stdstreamio sio;

const char *tree[] = {"willow", "pine", "fir", NULL};
tarray_tstring my_arr = tree;

my_arr.replace(1, 1, "linden", 1);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n", i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = willow
my_arr[1] = linden
my_arr[2] = fir
```

10.4.21 replace(), vreplace()

NAME

replace(), vreplace() — Replacement of elements (Specifies multiple strings or a string array)

SYNOPSIS

```
tarray_tstring &replace( size_t idx1, size_t n1,
                         const char *el0, const char *el1, ... );  ........ 1
tarray_tstring &vreplace( size_t idx1, size_t n1,
                          const char *el0, const char *el1, va_list ap );  2
tarray_tstring &replace( size_t idx1, size_t n1,
```

```
                          const char *const *elements );  ..................  3
tarray_tstring &replace( size_t idx1, size_t n1,
                          const char *const *elements, size_t n2 );  .......  4
tarray_tstring &replace( size_t idx1, size_t n1,
                          const tarray_tstring &src, size_t idx2 = 0 );  ..  5
tarray_tstring &replace( size_t idx1, size_t n1, const tarray_tstring &src,
                          size_t idx2, size_t n2 );  ........................  6
```

DESCRIPTION

Replaces `n1` elements starting from the element number `idx1` of a string array inside an object with the multiple strings specified by `el0, el1, ...` or the string array specified by `elements` and `src`.

If `idx1` has a value larger than the number of array elements specified to it the function performs the same processing as the `append()` member function (§10.4.16). If the sum of `idx1` and `n1` is larger than the number of elements of the array or the array needs to be expanded or contracted because of the size comparison between `n1` and `n2` the number of the elements is automatically adjusted.

Member functions 1 and 2 specify `el0`, `el1` and the variable-length argument or the list `ap` of variable-length arguments. Variable-length arguments must be `NULL`-terminated.

Member functions 3 and 4 specify to the argument `elements` a pointer array for a string. With member function 3 pointer arrays must be `NULL`-terminated. With member function 4 the number of elements can be specified by `n2`. If `n2` larger than the number of `elements` (until reaching `NULL`) specified then `n2` is ignored.

Member functions 5 and 6 enable `idx2` to be used to specify the element position to start the string array `src` to be sourced, and the number of elements by `n2`. Member function 5 can be used without specifying `idx2`. However, the function is processed as though `0` had been specified. Member function 6 enables the number `n2` of elements to be sourced to be specified. Please note that the element number for the first element of arrays is always `0`.

PARAMETER

[I]	`idx1`	Position to start an array inside an object
[I]	`n1`	Number of elements to be replaced
[I]	`el0`	String to be sourced (0th)
[I]	`el1`	String to be sourced (first)
[I]	`...`	String to be sourced (The second and following need to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for a string to be sourced (The second and following need to be `NULL`-terminated)
[I]	`elements`	Pointer array for a string to be sourced (With member function 3 must be `NULL`-terminated)
[I]	`n2`	Number of `elements` of the array, or the number of elements in `src` (When assigning a sub-array in `src`)
[I]	`src`	tarray_tstring class object that includes the string array to be sourced
[I]	`idx2`	Position to start an element in `src` (When assigning a sub-array in `src`)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code replaces 1 element starting from the element number 1 of the string array `my_tree` with two elements starting from the element number 1 of the string array `my_addTree`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring my_tree("willow", "pine", "fir", NULL);
tarray_tstring my_addTree("linden", "beech", "holly", NULL);

my_tree.replace(1, 1, my_addTree, 1, 2);
for ( size_t i = 0 ; i < my_tree.length() ; i++ ) {
    sio.printf("my_tree[%zu] = %s\n", i, my_tree.cstr(i));
}
```

Result of execution

```
my_tree[0] = willow
my_tree[1] = beech
my_tree[2] = holly
my_tree[3] = fir
```

10.4.22 erase()

NAME

erase() — Deletion of elements

SYNOPSIS

```
tarray_tstring &erase();  ..... 1
tarray_tstring &erase( size_t index, size_t num_elements = 1 );  ..... 2
```

DESCRIPTION

Deletes elements of a string array.

Member function 1 deletes all the array elements (The array length becomes zero).

Member function 2 deletes `num_elements` elements starting from the element with the element number `index`. Please note that the element number for the first element of arrays is always `0`. When `num_elements` is not specified, an element is deleted.

The array length decreases by the length deleted.

If `index` has a value larger than the length of an array specified to it the value is simply ignored.

PARAMETER

[I]	`index`	Element number
[I]	`num_elements`	Number of elements

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code deletes two elements from element number 1 of the string array `my_menu`, and then prints the result to standard output:

```
stdstreamio sio;

tarray_tstring my_menu("rice ball", "sushi", "tofu", NULL);
my_menu.erase(1,2);
for ( size_t i = 0 ; i < my_menu.length() ; i++ ) {
    sio.printf("my_menu[%zu] = %s\n", i, my_menu.cstr(i));
}
```

Result of execution

```
my_menu[0] = rice ball
```

10.4.23 clean()

NAME

clean() — Pads all the element values of an existing array with a string

SYNOPSIS

```
tarray_tstring &clean(const char *str = "");  ..... 1
tarray_tstring &clean(const tstring &str);  ..... 2
```

DESCRIPTION

Pads all the elements of a string array with the string `str`. The function can also be used without specifying the argument `str`. In that case, however, the function is processed as though the string `""` had been specified. Executing `clean()` does not change the length of a string array.

PARAMETER

[I] `str` String to pad a string array with

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code sets the value of `tree` to the string array object `my_arr`, and then sets `paulownia` to all the elements. It then prints the element values to standard output in order to verify them:

```
stdstreamio sio;

const char *tree[] = {"katsura", "torreya", NULL};
tarray_tstring my_arr = tree;

my_arr.clean("paulownia");
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n",i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = paulownia
my_arr[1] = paulownia
```

10.4.24 resize()

NAME

resize() — Changes the length of a string array

SYNOPSIS

```
tarray_tstring &resize( size_t new_num_elements );
```

DESCRIPTION

Changes the length of a string array to `new_num_elements`.

Increasing the length of the string array results in elements comprised of the empty string `""` being added to it.

Decreasing the length of the string array results in string elements after new_num_elements being deleted.

PARAMETER

[I] `new_num_elements` Length of string array after being changed
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code sets the value of `tree` to the string array object `my_arr`, and then changes the length of the string array to 2. It then prints the content of the element values to standard output in order to verify them:

```
stdstreamio sio;

const char *tree[] = {"andromeda", "yew", "Japanese pagoda tree", NULL};
tarray_tstring my_arr = tree;
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n",i, my_arr.cstr(i));
}

my_arr.resize(2);
for ( size_t i = 0 ; i < my_arr.length() ; i++ ) {
    sio.printf("my_arr[%zu] = %s\n",i, my_arr.cstr(i));
}
```

Result of execution

```
my_arr[0] = andromeda
my_arr[1] = yew
my_arr[2] = Japanese pagoda tree
```

```
my_arr[0] = andromeda
my_arr[1] = yew
```

10.4.25 resizeby()

NAME

resizeby() — Relatively changes the length of a string array

SYNOPSIS

```
tarray_tstring &resizeby( ssize_t len );
```

DESCRIPTION

Changes the length of a string array by as much as the length of `len`.

Increasing the length of the string array results in elements comprised of the empty string `""` being added to it.

Decreasing the length of a string array length results in the last abs(`len`) string elements being deleted.

PARAMETER

[I] `len` Increase/decrease in the length of an array
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.26 crop()

NAME

crop() — Cropping of string arrays

SYNOPSIS

```
tarray_tstring &crop( size_t idx, size_t len );
tarray_tstring &crop( size_t idx );
```

DESCRIPTION

Changes an array object to an array comprised of only `len` elements starting from the element number `idx`. If `len` is omitted the array will be comprised of only the elements in and after `idx`.

PARAMETER

[I] `idx` Position to start cropped element
[I] `len` Number of elements
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.27 chomp()

NAME

chomp() — Elimination of newline characters in all the elements

SYNOPSIS

```
tarray_tstring &chomp( const char *rs = "\n" );
tarray_tstring &chomp( const tstring &rs );
```

DESCRIPTION

Eliminates a newline character on the right end of all the elements of a string array.

This member function executes the chomp() member function (§9.5.25) for the tstring class on all the elements of an array. For more details refer to §9.5.25.

PARAMETER

[I] `rs` Newline character string

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.28 trim()

NAME

trim() — Elimination of spaces on both ends of all the elements

SYNOPSIS

```
tarray_tstring &trim( const char *side_spaces = " \t\n\r\f\v" );
tarray_tstring &trim( const tstring &side_spaces );
tarray_tstring &trim( int side_space );
```

DESCRIPTION

Eliminates arbitrary characters on both ends of a string in all the elements of a string array.

`side_spaces` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

This member function executes the trim() member function for the tstring class on all the elements of an array. For more details refer to §9.5.26.

PARAMETER

[I] `side_space` Arbitrary character

[I] `side_spaces` Arbitrary character string

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code divides a CSV-format string into elements using the split() member function (§10.4.12), assigns them as an array to an object, and then eliminates any unnecessary white space characters on the right and left ends of each element using `trim()`:

```
tarray_tstring my_arr;
my_arr.split(" MZ-2500, PC-8801MR2  ,FV77AV  ", ",", true);
my_arr.dprint();
my_arr.trim();
my_arr.dprint();
```

Result of execution

```
sli::tarray_tstring[obj=0x7fbffff470] = {" MZ-2500", " PC-8801MR2  ", "FV77AV  "}
sli::tarray_tstring[obj=0x7fbffff470] = {"MZ-2500", "PC-8801MR2", "FV77AV"}
```

10.4.29 ltrim()

NAME

ltrim() — Elimination of a space on the left end of all the elements

SYNOPSIS

```
tarray_tstring &ltrim( const char *side_spaces = " \t\n\r\f\v" );
tarray_tstring &ltrim( const tstring &side_spaces );
tarray_tstring &ltrim( int side_space );
```

DESCRIPTION

Eliminates an arbitrary character on the left end of a string in all the elements of a string array.

`side_spaces` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

This member function executes the ltrim() member function for the tstring class on all the elements of an array. For more details refer to §9.5.27.

PARAMETER

[I] `side_space` Arbitrary character
[I] `side_spaces` Arbitrary character string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.30 rtrim()

NAME

rtrim() — Elimination of a space on the right end of all the elements

SYNOPSIS

```
tarray_tstring &rtrim( const char *side_spaces = " \t\n\r\f\v" );
tarray_tstring &rtrim( const tstring &side_spaces );
tarray_tstring &rtrim( int side_space );
```

DESCRIPTION

Eliminates an arbitrary character on the right end of a string in all the elements of a string array.

`side_spaces` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

This member function executes the rtrim() member function for the tstring class on all the elements of an array. For more details refer to §9.5.28.

PARAMETER

[I] `side_space` Arbitrary character
[I] `side_spaces` Arbitrary character string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.31 strreplace()

NAME

strreplace() — String search and replacement of all the elements

SYNOPSIS

```
tarray_tstring &strreplace( const char *org_str, const char *new_str,
                            bool all = false );
tarray_tstring &strreplace( const tstring &org_str, const char *new_str,
                            bool all = false );
tarray_tstring &strreplace( const char *org_str, const tstring &new_str,
                            bool all = false );
tarray_tstring &strreplace( const tstring &org_str, const tstring &new_str,
                            bool all = false );
```

DESCRIPTION

Searches all the elements of a string array from the left side of a string for the string `org_str`, and if the string is found replaces it with the string `new_str`.

This member function executes the strreplace() member function for the tstring class on all the elements of an array (`0` is set to `pos`). For more details refer to §9.5.29.

PARAMETER

[I] `org_str` String to be detected
[I] `new_str` String to be sourced for replacement
[I] `all` Replace All flag
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code replaces the uppercase character version of N in all the elements with the lowercase version:

```
tarray_tstring my_arr("NEC", "NVIDIA", "HYNIX", NULL);
my_arr.strreplace("N", "n", true);
my_arr.dprint();
```

Result of execution

```
sli::tarray_tstring[obj=0x7fbffff470] = {"nEC", "nVIDIA", "HYnIX"}
```

10.4.32 regreplace()

NAME

regreplace() — String search and replacement on all the elements using a regular expression

SYNOPSIS

```
tarray_tstring &regreplace( const char *pat,
                            const char *new_str, bool all = false );
tarray_tstring &regreplace( const tstring &pat,
                            const char *new_str, bool all = false );
tarray_tstring &regreplace( const tregex &pat,
                            const char *new_str, bool all = false );
tarray_tstring &regreplace( const char *pat,
                            const tstring &new_str, bool all = false );
tarray_tstring &regreplace( const tstring &pat,
                            const tstring &new_str, bool all = false );
tarray_tstring &regreplace( const tregex &pat,
                            const tstring &new_str, bool all = false );
```

DESCRIPTION

Replaces with the string `new_str` parts all the elements of a string array that match the POSIX Extended Regular Expression (hereinafter referred to as a regular expression) specified by `pat`. Back references `"\\0"` through `"\\9"` can be used for `new_str` (`"\\0"` refers to an entire matching part). If you want to use the backslash specify `"\\\\"`.

This member function executes the regreplace() member function for the tstring class on all the elements of an array (`0` is set to `pos`). For more details refer to §9.5.30.

If you do not need to use a regular expression the strreplace() member function (§10.4.31), which operates at higher speed, can be used.

PARAMETER

[I]	`pat`	Character pattern (regular expression) or compiled object for the tregex class
[I]	`new_str`	String after the replacement
[I]	`all`	Replace All flag

([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXCEPTION
If the system failed to secure an internal buffer.

EXAMPLE
The following code replaces the parts of all the elements that match `"[- ]"` with an underscore:

```
tarray_tstring my_arr("MZ-2000", "PC-88VA", "X1 turboZ III", NULL);
my_arr.regreplace("[- ]", "_", true);
my_arr.dprint();
```

Result of execution
```
sli::tarray_tstring[obj=0x7fbffff470] = {"MZ_2000", "PC_88VA", "X1_turboZ_III"}
```

10.4.33 tolower()

NAME
tolower() — Replaces the uppercase version of characters in all the elements with the lowercase version

SYNOPSIS
```
tarray_tstring &tolower();
```

DESCRIPTION
Replaces the uppercase version of alphabetical characters in all the elements of a string array with the lowercase version.

This member function executes the tolower() member function for the tstring class on all the elements of an array. For more details refer to §9.5.31.

RETURN VALUE
Reference to itself

10.4.34 toupper()

NAME
toupper() — Replaces the lowercase version of characters in all the elements with the uppercase version

SYNOPSIS
```
tarray_tstring &toupper();
```

DESCRIPTION
Replaces the lowercase version of alphabetical characters in all the elements of a string array with the uppercase version.

This member function executes the toupper() member function for the tstring class on all the elements of an array. For more details refer to §9.5.32.

RETURN VALUE
Reference to itself

10.4.35 expand_tabs()

NAME

expand_tabs() — Replaces TAB characters in all the elements with white space characters

SYNOPSIS

```
tarray_tstring &expand_tabs( size_t tab_width = 8 );
```

DESCRIPTION

Replaces horizontal tabulation characters `'\t'` in all the elements of a string array with white space characters, tabulating to the value of `tab_width`.

This member function executes the expand_tabs() member function for the tstring class on all the elements of an array. For more details refer to §9.5.33.

PARAMETER

[I] `tab_width` A TAB width
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.36 contract_spaces()

NAME

contract_spaces() — Replaces white space characters in all the elements with TAB characters

SYNOPSIS

```
tarray_tstring &contract_spaces( size_t tab_width = 8 );
```

DESCRIPTION

Replaces with `'\t'` all occurrences of two or more contiguous white space characters `' '` in all the elements of a string array that tabulate to the specified TAB width of `tab_width`.

This member function executes the contract_spaces() member function for the tstring class on all the elements of an array. For more details §9.5.34.

PARAMETER

[I] `tab_width` A TAB width
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

10.4.37 find_elem()

NAME

find_elem() — Searches from the left side (start) for an array element

SYNOPSIS

```
ssize_t find_elem( const char *str ) const;
ssize_t find_elem( size_t idx, const char *str ) const;
ssize_t find_elem( size_t idx, const char *str, size_t *nextidx ) const;
ssize_t find_elem( const tstring &str ) const;
ssize_t find_elem( size_t idx, const tstring &str ) const;
ssize_t find_elem( size_t idx, const tstring &str, size_t *nextidx ) const;
```

DESCRIPTION

Searches array elements from the left side for an element that exactly matches the string `str`, and if an element is found returns the element number for the element but if no element is found returns a negative number.

If you want the search to start from a specific element the start position can be specified using the argument `idx`. Please note that the element number for the first element of arrays is always `0`.

If you want to continuously search for elements `nextidx` can be used to acquire the value that should be provided to `idx` in the next iteration. For the variable referred to by `nextidx`, if an element is found, the position one element to the right of the position in which the element was found is returned, but if no element is found the length of an array of the function itself. If you do not need to acquire a value using `nextidx` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`str`	String that matches an element value to be detected
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If an element that matches `str` is found the element number of the element.
Negative value (Error)	:	If no element that matches `str` is found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `str` is `NULL`.

EXAMPLE

The following code searches the string array `my_arr`, and then lists the elements that match `"INTEL"` in the order of from the beginning of the array. `idx` provides the element number from which to start detection, and these addresses are provided to the last value for the find_elem() member function to ensure that the appropriate values are automatically used:

```
stdstreamio sio;
tarray_tstring my_arr;
size_t idx = 0;
ssize_t fidx;
my_arr.assign("ZILOG", "INTEL", "INTEL", "MOTOROLA", "MOS", NULL);
```

```
while ( 0 <= (fidx=my_arr.find_elem(idx, "INTEL", &idx)) ) {
    sio.printf("in : fidx=%zd nextidx=%zu\n", fidx, idx);
}
sio.printf("out: fidx=%zd nextidx=%zu\n", fidx, idx);
```

Result of execution

```
in : fidx=1 nextidx=2
in : fidx=2 nextidx=3
out: fidx=-1 nextidx=5
```

10.4.38 rfind_elem()

NAME

rfind_elem() — Searches from the right side (end) for an array element

SYNOPSIS

```
ssize_t rfind_elem( const char *str ) const;
ssize_t rfind_elem( size_t idx, const char *str ) const;
ssize_t rfind_elem( size_t idx, const char *str, size_t *nextidx ) const;
ssize_t rfind_elem( const tstring &str ) const;
ssize_t rfind_elem( size_t idx, const tstring &str ) const;
ssize_t rfind_elem( size_t idx, const tstring &str, size_t *nextidx ) const;
```

DESCRIPTION

Searches array elements from the right side for an element that exactly matches the string `str`, and if the element is found, returns the element number for the element but if no element is found returns a negative number.

If you want the search to start from a specific element the start position can be specified using the argument `idx`. Please note that the element number for the first element of arrays is always `0`.

If you want to continuously search for elements `nextidx` can be used to acquire the value that should be provided to `idx` in the next iteration. With the variable referred to by `nextidx`, if an element is found when `idx` is 1 or more the position one element to the left of the position in which the element was found is returned, but otherwise the length of the array of the function itself is returned. If you do not need to acquire a value using `nextidx` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`str`	String that matches an element value to be detected
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If an element that matches `str` is found the element number of the element.
Negative value (Error)	:	If no element that matches `str` is found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `str` is `NULL`.

EXAMPLE

The following code searches the string array `my_arr`, and then lists the elements that match `"INTEL"` in the order of from the end of the array. `idx` is the element number from which to start detection, and these addresses are provided to the last value for the rfind_elem() member function to ensure that the appropriate values are automatically used:

```
stdstreamio sio;
tarray_tstring my_arr;
size_t idx;
ssize_t fidx;
my_arr.assign("ZILOG", "INTEL", "INTEL", "MOTOROLA", "MOS", NULL);
idx = my_arr.length() - 1;
while ( 0 <= (fidx=my_arr.rfind_elem(idx, "INTEL", &idx)) ) {
    sio.printf("in : fidx=%zd nextidx=%zu\n", fidx, idx);
}
sio.printf("out: fidx=%zd nextidx=%zu\n", fidx, idx);
```

Result of execution

```
in : fidx=2 nextidx=1
in : fidx=1 nextidx=0
out: fidx=-1 nextidx=5
```

10.4.39 find()

NAME

find() — Searches an array from the left side (start) for a string

SYNOPSIS

```
ssize_t find( const char *str, ssize_t *pos_r ) const;
ssize_t find( size_t idx, size_t pos, const char *str,
              ssize_t *pos_r ) const;
ssize_t find( size_t idx, size_t pos, const char *str,
              ssize_t *pos_r,  size_t *nextidx, size_t *nextpos ) const;
ssize_t find( const tstring &str, ssize_t *pos_r ) const;
ssize_t find( size_t idx, size_t pos, const tstring &str,
              ssize_t *pos_r ) const;
ssize_t find( size_t idx, size_t pos, const tstring &str,
              ssize_t *pos_r, size_t *nextidx, size_t *nextpos ) const;
```

DESCRIPTION

Searches array elements from the left side for an element that includes the string `str`, and if an element is found returns the element number of the element but if no element is found returns a negative number as the return value for the member function. If an element is found it also concurrently returns the position of the string in that element to a variable referred to by `pos_r`.

If you want the search to start from the position of a specific string in a specific element the start position can be specified using the arguments `idx` and `pos`. Please note that the element number for the first element both in arrays and strings is always `0`. If the arguments `idx` and `pos` are omitted the search will start from the beginning of the strings in an element with the element number 0.

If you want to continuously search for elements `nextidx` and `nextpos` can be used to acquire the values that should be provided to `idx` and `pos` in the next iteration. These values can be more easily understood by examining some example code than by reading a text description. Refer to EXAMPLE provided below.

If you do not need to acquire values using `pos_r`, `nextidx` and `nextpos` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pos`	Position to start searching for a string
[I]	`str`	String to be detected
[O]	`pos_r`	If an element is found the position of the string in the element
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If the string specified is found the element number of the element.
Negative value (Error)	:	If a string specified is not found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pos` has a value larger than the length of a string specified to it.
	:	If `str` is `NULL`.

EXAMPLE

The following code searches the string array `my_arr`, and the lists the parts that include `"80"`in the order of from the beginning of the array. `idx` and `pos` are the element number and string position from which to start detection, and these addresses are provided to the last two values for the find() member function to ensure that the appropriate values are automatically used:

```
stdstreamio sio;
tarray_tstring my_arr;
size_t idx = 0, pos = 0;
ssize_t fidx, fpos;
my_arr.assign("Z80", "8080", "8086", "6800", "6502", NULL);
while ( 0 <= (fidx=my_arr.find(idx, pos, "80", &fpos, &idx, &pos)) ) {
    sio.printf("in : fidx=%zd fpos=%zd nextidx=%zu nextpos=%zu\n",
               fidx, fpos, idx, pos);
}
sio.printf("out: fidx=%zd fpos=%zd nextidx=%zu nextpos=%zu\n",
           fidx, fpos, idx, pos);
```

Result of execution

```
in : fidx=0 fpos=1 nextidx=0 nextpos=3
in : fidx=1 fpos=0 nextidx=1 nextpos=2
in : fidx=1 fpos=2 nextidx=1 nextpos=4
in : fidx=2 fpos=0 nextidx=2 nextpos=2
```

```
in : fidx=3 fpos=1 nextidx=3 nextpos=3
out: fidx=-1 fpos=-1 nextidx=5 nextpos=5
```

10.4.40 rfind()

NAME

rfind() — Searches an array from the right side (end) for a string

SYNOPSIS

```
ssize_t rfind( const char *str, ssize_t *pos_r ) const;
ssize_t rfind( size_t idx, size_t pos, const char *str,
               ssize_t *pos_r ) const;
ssize_t rfind( size_t idx, size_t pos, const char *str,
               ssize_t *pos_r,  size_t *nextidx, size_t *nextpos ) const;
ssize_t rfind( const tstring &str, ssize_t *pos_r ) const;
ssize_t rfind( size_t idx, size_t pos, const tstring &str,
               ssize_t *pos_r ) const;
ssize_t rfind( size_t idx, size_t pos, const tstring &str,
               ssize_t *pos_r, size_t *nextidx, size_t *nextpos ) const;
```

DESCRIPTION

Searches array elements from the right side for an element that includes the string `str`, and if the element is found returns the element number of the element but if no element is found returns a negative number as the return value for the member function. If an element is found it also concurrently returns the position of the string in that element to the variable referred to by `pos_r`.

If you want the search to start from the position of a specific string in a specific element the start position can be specified using the arguments `idx` and `pos`. Please note that the element number for the first element (on the left end) both in arrays and strings is always `0`. If the arguments `idx` and `pos` are omitted the search will start from the end (string length) of the strings in the last element (number of elements − 1).

If you want to continuously search for elements `nextidx` and `nextpos` can be used to acquire the values that should be provided to `idx` and `pos` in the next iteration. These values can be more easily understood by examining some example code than by reading a text description. Refer to the EXAMPLE provided below.

If you do not need to acquire values using `pos_r`, `nextidx` and `nextpos` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pos`	Position to start searching for a string
[I]	`str`	String to be detected
[O]	`pos_r`	If an element is found the position of the string in the element
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If the string specified is found the element number of the element.
Negative value (Error)	:	If a string that is specified is not found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pos` has a value larger than the length of a string specified to it.
	:	If `str` is `NULL`.

EXAMPLE

The following code searches the string array `my_arr`, and then lists parts that include `"80"` in the order of from the end of the array. `idx` and `pos` are the element number and string position from which to start detection, and these addresses are provided to the last two values for the rfind() member function to ensure that the appropriate values are automatically used:

```
stdstreamio sio;
tarray_tstring my_arr;
size_t idx, pos;
ssize_t fidx, fpos;
my_arr.assign("Z80", "8080", "8086", "6800", "6502", NULL);
idx = my_arr.length() - 1;
pos = my_arr.length(idx);
while ( 0 <= (fidx=my_arr.rfind(idx, pos, "80", &fpos, &idx, &pos)) ) {
    sio.printf("in : fidx=%zd fpos=%zd nextidx=%zu nextpos=%zu\n",
               fidx, fpos, idx, pos);
}
sio.printf("out: fidx=%zd fpos=%zd nextidx=%zu nextpos=%zu\n",
           fidx, fpos, idx, pos);
```

Result of execution

```
in : fidx=3 fpos=1 nextidx=2 nextpos=4
in : fidx=2 fpos=0 nextidx=1 nextpos=4
in : fidx=1 fpos=2 nextidx=1 nextpos=0
in : fidx=1 fpos=0 nextidx=0 nextpos=3
in : fidx=0 fpos=1 nextidx=5 nextpos=4
out: fidx=-1 fpos=-1 nextidx=5 nextpos=4
```

10.4.41 find_matched_str()

NAME

find_matched_str() — Searches for an element (string) that matches a pattern

SYNOPSIS

```
ssize_t find_matched_str( const char *pat ) const;
ssize_t find_matched_str( size_t idx, const char *pat ) const;
ssize_t find_matched_str( size_t idx, const char *pat, size_t *nextidx ) const;
ssize_t find_matched_str( const tstring &pat ) const;
ssize_t find_matched_str( size_t idx, const tstring &pat ) const;
ssize_t find_matched_str( size_t idx, const tstring &pat, size_t *nextidx ) const;
```

DESCRIPTION

Attempts string matching on array elements in the order of from the left side using Shell's wild card patterns, and if an element matches the pattern returns the element number.

If you need to treat a period '.' at the beginning of a string or a slash '/' in a special manner use the find_matched_fn() member function (§10.4.42) or the find_matched_pn() member function (§10.4.43).

This member function executes the strmatch() member function for the tstring class on all the elements of an array in order (`0` is set to `pos`). For more details refer to §9.5.58.

If you want the search to start from a specific element the start position can be specified using the argument `idx`. Please note that the element number for the first element of arrays is always `0`.

If you want to continuously search for elements `nextidx` can be used to acquire the value that should be provided to `idx` in the next iteration. With the variable referred to by `nextidx`, if an element that matches is found the position one element to the right of the position in which the element was found is returned but if no element is found the length of the array of the function itself is returned. If you do not need to acquire a value using `nextidx` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pat`	String that matches an element value to be detected
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If an element that matches is found the element number of the element.
Negative value (Error)	:	If no element that matches is found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pat` is `NULL`.

EXAMPLE

The following code searches the string array `my_arr`, and then lists elements that match `"MO*"` in the order of from the beginning of the array. `idx` is the element number from which to start detection, and these addresses are provided to the last value for the find_matched_str() member function to ensure that the appropriate values are automatically used:

```
stdstreamio sio;
tarray_tstring my_arr;
size_t idx = 0;
ssize_t fidx;
my_arr.assign("ZILOG", "INTEL", "MOTOROLA", "MOS", "AMD", NULL);
while ( 0 <= (fidx=my_arr.find_matched_str(idx, "MO*", &idx)) ) {
    sio.printf("in : fidx=%zd nextidx=%zu\n", fidx, idx);
}
sio.printf("out: fidx=%zd nextidx=%zu\n", fidx, idx);
```

Result of execution

```
in : fidx=2 nextidx=3
```

```
in : fidx=3 nextidx=4
out: fidx=-1 nextidx=5
```

10.4.42 find_matched_fn()

NAME

find_matched_fn() — Searches for an element (file name) that matches a pattern

SYNOPSIS

```
ssize_t find_matched_fn( const char *pat ) const;
ssize_t find_matched_fn( size_t idx, const char *pat ) const;
ssize_t find_matched_fn( size_t idx, const char *pat, size_t *nextidx ) const;
ssize_t find_matched_fn( const tstring &pat ) const;
ssize_t find_matched_fn( size_t idx, const tstring &pat ) const;
ssize_t find_matched_fn( size_t idx, const tstring &pat, size_t *nextidx ) const;
```

DESCRIPTION

Attempts string matching on array elements in the order of from the left side using Shell's wild card patterns, and if an element matches the pattern returns the element number.

The find_matched_fn() function is assumed to be used in searches for file names, and hence treats the period `'.'` at the beginning of a string in a special manner. In other words, the function disables the wild cards `'*'` and `'?'` to match the period `'.'` at the beginning of a string.

If you need to treat a slash `'/'` in a special manner use the find_matched_pn() member function (§10.4.43).

This member function executes the fnmatch() member function for the tstring class on all the elements of an array in order (`0` is set to `pos`). For more details refer to §9.5.58.

If you want the search to start from a specific element the start position can be specified using the argument `idx`. Please note that the element number for the first element of arrays is always `0`.

If you want to continuously search for elements `nextidx` can be used to acquire the value that should be provided to `idx` in the next iteration. With the variable referred to by `nextidx`, if an element that matches is found the position one element to the right of the position in which the element was found is returned but if no element is found the length of the array of the function itself is returned. If you do not need to acquire a value using `nextidx` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pat`	String that matches an element value to be detected
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If an element that matches is found the element number of the element.
Negative value (Error)	:	If no element that matches is found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pat` is `NULL`.

EXAMPLE

Refer to the EXAMPLE in §10.4.41.

10.4.43 find_matched_pn()

NAME

find_matched_pn() — Searches for an element (path name) that matches a pattern

SYNOPSIS

```
ssize_t find_matched_pn( const char *pat ) const;
ssize_t find_matched_pn( size_t idx, const char *pat ) const;
ssize_t find_matched_pn( size_t idx, const char *pat, size_t *nextidx ) const;
ssize_t find_matched_pn( const tstring &pat ) const;
ssize_t find_matched_pn( size_t idx, const tstring &pat ) const;
ssize_t find_matched_pn( size_t idx, const tstring &pat, size_t *nextidx ) const;
```

DESCRIPTION

Attempts string matching on array elements in the order of from the left side using Shell's wild card patterns, and if an element matches the pattern returns the element number.

The find_matched_pn() member function is assumed to be used in searches for path names, and hence treats the period `'.'` at the beginning of a string, a slash `'/'` and the period that immediately follows a slash in a special manner. The function disables the wild cards `'*'` and `'?'` to match these characters.

This member function executes the pnmatch() member function for the tstring class on all the elements of an array in order (`0` is set to `pos`). For more details refer to §9.5.58.

If you want the search to start from a specific element the start position can be specified using the argument `idx`. Please note that the element number for the first element of arrays is always `0`.

If you want to continuously search for elements `nextidx` can be used to acquire the value that should be provided to `idx` in the next iteration. With the variable referred to by `nextidx`, if an element that matches is found the position one element to the right of the position in which the element was found is returned but if no element is found the length of the array in the function is returned. If you do not need to acquire a value using `nextidx` NULL can also be used.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pat`	String that matches an element value to be detected
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If an element that matches is found the element number of the element.
Negative value (Error)	:	If no element that matches is found.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pat` is `NULL`.

EXAMPLE

Refer to the EXAMPLE in §10.4.41.

10.4.44 regmatch() [Normal edition]

NAME

regmatch() — Searches for a string using an extended regular expression

SYNOPSIS

```
ssize_t regmatch( const char *pat, ssize_t *pos_r,
                  size_t *span_r ) const;  ..................................... 1
ssize_t regmatch( size_t idx, size_t pos, const char *pat,
                  ssize_t *pos_r, size_t *span_r ) const;  ..................... 2
ssize_t regmatch( size_t idx, size_t pos, const char *pat,
                  ssize_t *pos_r, size_t *span_r,
                  size_t *nextidx, size_t *nextpos ) const;  ................... 3
ssize_t regmatch( const tstring &pat, ssize_t *pos_r,
                  size_t *span_r ) const;  ..................................... 4
ssize_t regmatch( size_t idx, size_t pos, const tstring &pat,
                  ssize_t *pos_r, size_t *span_r ) const;  ..................... 5
ssize_t regmatch( size_t idx, size_t pos, const tstring &pat,
                  ssize_t *pos_r, size_t *span_r,
                  size_t *nextidx, size_t *nextpos ) const;  ................... 6
ssize_t regmatch( const tregex &pat, ssize_t *pos_r,
                  size_t *span_r ) const;  ..................................... 7
ssize_t regmatch( size_t idx, size_t pos, const tregex &pat,
                  ssize_t *pos_r, size_t *span_r ) const;  ..................... 8
ssize_t regmatch( size_t idx, size_t pos, const tregex &pat,
                  ssize_t *pos_r, size_t *span_r,
                  size_t *nextidx, size_t *nextpos ) const;  ................... 9
```

DESCRIPTION

Searches array elements from the left side for an element that includes the part that matches the POSIX Extended Regular Expression (hereinafter referred to as a regular expression) specified by `pat`, and if a part matches the expression returns the element number but if no part matches the expression returns a negative number as the return value for the member function. If a part matches the expression it also concurrently returns to the variables referred to by `pos_r` and `span_r` the character position and length of the part of the element that matches the expression.

If you want the search to start from the position of a specific string in a specific element the start position can be specified using the arguments `idx` and `pos`. Please note that the element number for the first element both in arrays and strings is always `0`. If the arguments `idx` and `pos` are omitted the search will start from the beginning of strings in an element with the element number 0.

Member functions 1 to 6 compile the regular expression `pat`, save the result to an internal buffer that the functions encompass, and then perform matching (If `pat` is the same as the one previously compiled it is not recompiled again).

Member functions 7 to 9 specify an object for the tregex class that will retain the result of compiling the regular expression. The regular expression will therefore need to be compiled in advance using the compile() member function for the tregex class before use of the regmatch() member function (Refer to the EXAMPLE).

In both cases if the function fails to compile the regular expression it outputs the content to standard error output.

If you want to continuously search for elements `nextidx` and `nextpos` can be used to acquire the values that should be provided to `idx` and `pos` in the next iteration. These values can be more easily understood by examining some example of the code than by reading a text description. Refer to the EXAMPLE provided below.

If you do not need to acquire values using `pos_r`, `nextidx` and `nextpos` NULL can also be used.

For more details on regular expressions refer to §9.5.59.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pos`	Position to start searching for a string
[I]	`pat`	Regular expression to be used in search
[O]	`pos_r`	If a part matches the expression the character position of the part of that element that matches the expression.
[O]	`span_r`	If a part matches the expression the character length of the part of that element that matches the expression.
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a specified regular expression matches a part the element number of the element.
Negative value (Error)	:	If a specified regular expression does not match any part.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pos` has a value larger than the length of a string specified to it.
	:	If a regular expression specified by `pat` is invalid (For more details refer to §9.5.59).

EXAMPLE

The following code searches the string array `my_arr`, and then lists the parts that include the regular expression `"http://[^/]+\\.jp/"` in the order of from the beginning of the array. `idx` and `pos` are the element number and string position from which to start detection, and these addresses are provided to the last two values for the regmatch() member function to ensure that appropriate values are automatically used:

```
stdstreamio sio;
tarray_tstring my_arr("http://www.jaxa.jp/", "http://www.noao.edu/", NULL);
size_t fspan, idx = 0, pos = 0;
ssize_t fidx, fpos;
tregex pat;
pat.compile("http://[^/]+\\.jp/");
while ( 0 <= (fidx=my_arr.regmatch(idx,pos,pat, &fpos,&fspan,&idx,&pos)) ) {
    sio.printf("in : fidx=%zd fpos=%zd fspan=%zd nextidx=%zu nextpos=%zu\n",
               fidx, fpos, fspan, idx, pos);
}
```

```
    sio.printf("out: fidx=%zd fpos=%zd fspan=%zd nextidx=%zu nextpos=%zu\n",
               fidx, fpos, fspan, idx, pos);
    pat.init();
```

Result of execution

```
in : fidx=0 fpos=0 fspan=19 nextidx=0 nextpos=19
out: fidx=-1 fpos=-1 fspan=0 nextidx=2 nextpos=21
```

10.4.45 regmatch() [Advanced edition]

NAME

regmatch() — Searches for a string using an extended regular expression

SYNOPSIS

```
ssize_t regmatch( const char *pat, tarray_tstring *result );  ..... 1
ssize_t regmatch( size_t idx, size_t pos, const char *pat,
                  tarray_tstring *result );  ..... 2
ssize_t regmatch( size_t idx, size_t pos, const char *pat,
                  tarray_tstring *result,
                  size_t *nextidx, size_t *nextpos );  ..... 3
ssize_t regmatch( const tstring &pat, tarray_tstring *result );  ..... 4
ssize_t regmatch( size_t idx, size_t pos, const tstring &pat,
                  tarray_tstring *result );  ..... 5
ssize_t regmatch( size_t idx, size_t pos, const tstring &pat,
                  tarray_tstring *result,
                  size_t *nextidx, size_t *nextpos );  ..... 6
ssize_t regmatch( const tregex &pat, tarray_tstring *result ) const;  ..... 7
ssize_t regmatch( size_t idx, size_t pos, const tregex &pat,
                  tarray_tstring *result ) const;  ..... 8
ssize_t regmatch( size_t idx, size_t pos, const tregex &pat,
                  tarray_tstring *result,
                  size_t *nextidx, size_t *nextpos ) const;  ..... 9
```

DESCRIPTION

Searches array elements from the left side for an element that includes the part that matches the POSIX Extended Regular Expression (hereinafter referred to as a regular expression) specified by `pat`, and if a part matches the expression returns the element number but if no part matches the expression returns a negative number as the return value for the member function. If a part matches the expression it also concurrently returns to the string array object `result` information that includes the back reference for the part of the element matching the expression.

These member functions perform regular expression matching using the regassign() member function for the argument object `result`. The method of acquiring the result information on the part that matches the expression is therefore the same as when the regassign() member function is used. For more details on the method refer to §10.4.13.

If you want the search to start from the position of a specific string in a specific element the start position can be specified using the arguments `idx` and `pos`. Please note that the element number for the first element both in arrays and strings is always `0`. If the arguments `idx` and `pos` are omitted the search will start from the beginning of strings in an element with the element number 0.

Member functions 1 to 6 compile the regular expression `pat`, save the result to an internal buffer that the functions encompass, and then perform the matching (If `pat` is the same as the one previously compiled it is not recompiled again).

Member functions 7 to 9 specify an object for the tregex class that retains the result of compiling the regular expression. The regular expression therefore needs to be compiled in advance using the compile() member function for the tregex class before use of the regmatch() member function (Refer to the EXAMPLE).

In both cases if the function fails to compile the regular expression it outputs the content to standard error output.

If you want to continuously search for elements`nextidx` and `nextpos` can be used to acquire the values that should be provided to `idx` and `pos` in the next iteration. These values can be more easily understood by examining some example code than by reading a text description. For more details on regular expressions refer to §10.4.44.

If you do not need to acquire values using `nextidx` and `nextpos` NULL can also be used.

For more details on regular expressions refer to §9.5.59.

PARAMETER

[I]	`idx`	Position to start searching for an array element
[I]	`pos`	Position to start searching for a string
[I]	`pat`	Regular expression to be used in search
[O]	`result`	If a part matches the expression the result information on the part of the element that matches the expression.
[O]	`nextidx`	`idx` for use in next search (Used in continuous searches)
[O]	`nextpos`	`pos` for use in next search (Used in continuous searches)

([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a specified regular expression matches a part the element number of the element.
Negative value (Error)	:	If a specified regular expression does not match any part.
	:	If there is no string inside an object.
	:	If `idx` has a value larger than the length of an array specified to it.
	:	If `pos` has a value larger than the length of a string specified to it.
	:	If a regular expression specified by `pat` is invalid (For more details refer to §9.5.59).
	:	If `result` is NULL.
	:	If the function itself is specified to `result`.

EXAMPLE-1

The following code searches the string array `my_arr` for an element that matches the regular expression `"^([ ]*)([^= ]+)([ ]*=[ ]*)([^= ]*)"`, and if an element is found then displays the back reference elements 2 and 4 as the key and value respectively:

```
stdstreamio sio;
tarray_tstring my_arr("HEADER",
                      " ARCHITECTURE = x86_64 / CPU = AMD", "OS = Linux  ",
                      NULL);
tarray_tstring my_result;
```

```
size_t pos = 0, idx = 0;
tregex pat;
pat.compile("^([ ]*)([^= ]+)([ ]*=[ ]*)([^= ]*)");
while ( 0 <= my_arr.regmatch(idx, pos, pat, &my_result, &idx, &pos) ) {
    if ( my_result.length() == 5 ) {
        sio.printf("key=[%s] value=[%s]\n",
                   my_result.cstr(2), my_result.cstr(4));
    }
}
pat.init();
```

Result of execution

```
key=[ARCHITECTURE] value=[x86_64]
key=[OS] value=[Linux]
```

EXAMPLE-2

The following code also has the same result as in EXAMPLE-1, but in this example the regular expression "^" is not used and instead the method of retrieving the "part of each element that first matches the expression". The code in EXAMPLE 1 is safer to use.

```
stdstreamio sio;
tarray_tstring my_arr("HEADER",
                      " ARCHITECTURE = x86_64 / CPU = AMD", "OS = Linux  ",
                      NULL);
tarray_tstring my_result;
ssize_t i;
tregex pat;
pat.compile("([^= ]+)([ ]*=[ ]*)([^= ]*)");
for ( i=0 ; 0 <= (i=my_arr.regmatch(i, 0, pat, &my_result)) ; i++ ) {
    if ( my_result.length() == 4 ) {
        sio.printf("key=[%s] value=[%s]\n",
                   my_result.cstr(1), my_result.cstr(3));
    }
}
pat.init();
```

11 ASARRAY_TSTRING class

The asarray_tstring class enables users to handle associative arrays of strings more easily. The class has the implementation requirement that normal string arrays have to be indexed in ensuring that its associative arrays will provide both high-speed read access along with the benefits that normal arrays provide.

tarray_tstring class (§10)) is used inside objects to manage keys and values, and can be combined with tarray_tstring class and tstring class (§9) APIs to provide easily used string array APIs.

The class has the following characteristics:

- Memory is automatically secured and hence objects can be assigned immediately after being created.
- The `printf()` notation can be used with many of the member functions.
- A wealth of other tstring class member functions are available for use through `[]`, the at() member function and the atf() member function.
- Can be used to easily divide space-delimited, TAB-delimited or CSV-format strings.
- Keys are managed by a dictionaryand hence the values can be retrieved very fast.
- Member functions are available that enable users to edit the all the elements of strings in arrays in a single stroke (e.g., `chomp()`, `trim()`, etc.). The functions can be used in the same manner as with the tstring class (§9).
- The keys() member function (§11.4.8) and values() member function (§11.4.9) make the search processing APIs (regular expressions etc) that use the tarray_tstring class available for use with internal key arrays and value arrays.

If you use the asarray_tstring class you must add "`#include <sli/asarray_tstring.h>`" to the code. In addition, if you need to declare a namespace (§4.1)) you must also add "`using namespace sli;`" to the code.

The following is a simple example of using the class.

```
#include <sli/stdstreamio.h>
#include <sli/asarray_tstring.h>
using namespace sli;

int main()
{
    stdstreamio sio;
    asarray_tstring my_aarr;
    /* Assign "RedHat" with a key as "VENDOR" */
    my_aarr["VENDOR"] = "RedHat";
    /* Assign "Linux" with a key as "OS" */
    my_aarr["OS"] = "Linux";
    /* Assign "2", "4" and "30" with keys as "VERSION_0", "VERSION_1" and "VERSION_2" */
    size_t i=0;
    my_aarr.atf("VERSION_%d",i++).printf("2");
    my_aarr.atf("VERSION_%d",i++).printf("4");
    my_aarr.atf("VERSION_%d",i++).printf("30");
    /* Display all the elements */
    for ( i=0 ; i < my_aarr.length() ; i++ ) {
        const char *key = my_aarr.key(i);
        sio.printf("%s ... [%s]\n", key, my_aarr.cstr(key));
    }

    return 0;
}
```

Result of execution

```
VENDOR ... [RedHat]
OS ... [Linux]
VERSION_0 ... [2]
VERSION_1 ... [4]
VERSION_2 ... [30]
```

11.1 Creating objects

There are the three methods of providing objects with an initial value[13].

With the first method no arguments are specified.

```
    asarray_tstring my_arr0;
```

In this situation neither a buffer for the string nor a buffer for the pointer array is secured.

The second method provides objects with an initial value using a variable-length argument.

```
    asarray_tstring my_arr0("OS","Solaris", "VENDOR","Sun", NULL);
```

In this case an associative array is initialized using the key and value strings provided. The arguments are provided in the order of key string 0, value string 0, key string 1, value string1 ... The end of the arguments must always be `NULL`.

The third method provides an array of the `asarrdef_tstring` structure type. The following is an example of this:

```
    asarrdef_tstring my_def0[] = { {"OS","Solaris"}, {"VENDOR","Sun"},
                                   {NULL,NULL} };
    asarray_tstring my_arr0(my_def0);
```

The last element of an array for the structure must always be {`NULL,NULL`}.

[13] The class does not include the operating modes the tstring class (§9) does.

11.2 List of member functions

Table 23 lists the member functions.

	Name of member function	Feature
§11.3.1	`[]`	Reference to the element value object (tstring class) corresponding to a specified key
§11.3.2	`=`	Copies objects
§11.4.1	`length()`	Length of associative array (number of arrays), and length of value string
§11.4.2	`cstrarray()`	Pointer array (NULL-terminated) for a value string
§11.4.3	`cstr()`, `cstrf()`	Value string corresponding to a specified key or element number
§11.4.4	`at()`, `atf()`	Reference to the element value object (tstring class) corresponding to a specified key or element number
§11.4.5	`at_cs()`, `atf_cs()`	Reference to the element value object (tstring class) corresponding to a specified key or element number (Read only)
§11.4.6	`index()`	Acquires the element number corresponding to a key string
§11.4.7	`key()`	Acquires the key string corresponding to an element number
§11.4.8	`keys()`	References the array object for a key string (Read only)
§11.4.9	`values()`	References the array object for a value string (Read only)

Table 23: List of member functions available for use with the asarray_tstring class (Continued on next page)

	Name of member function	Feature
§11.4.10	`dprint()`	Outputs object information to standard error output (For use debugging user programs)
§11.4.11	`swap()`	Interchanges objects
§11.4.12	`init()`	Complete initialization of objects
§11.4.13	`assign()`, `assignf()`	Initialization of objects and assignment of elements (Specifies a single set of a key and a value)
§11.4.14	`assign()`, `vassign()`	Initialization of objects and assignment of elements (Specifies multiple sets of a key and a value)
§11.4.15	`assign_keys()`	Sets multiple strings or a string array to keys
§11.4.16	`assign_values()`	Sets multiple strings or a string array to values
§11.4.17	`split_keys()`	Divides strings and sets them to keys
§11.4.18	`split_values()`	Divides strings and sets them to values
§11.4.19	`append()`, `appendf()`	Adds elements (Specifies a single set of a key and a value)
§11.4.20	`append()`, `vappend()`	Adds elements (Specifies multiple sets of a key and a value)
§11.4.21	`insert()`, `insertf()`	Inserts elements (Specifies a single set of a key and a value)
§11.4.22	`insert()`, `vinsert()`	Inserts elements (Specifies multiple sets of a key and a value)
§11.4.23	`erase()`	Deletes elements
§11.4.24	`clean()`	Pads all the element values of an existing associative array with any string
§11.4.25	`rename_a_key()`	Changes of key strings
§11.4.26	`chomp()`	Elimination of newline characters in all the elements
§11.4.27	`trim()`	Elimination of spaces on both ends of all the elements
§11.4.28	`ltrim()`	Elimination of a space on the left end of all the elements
§11.4.29	`rtrim()`	Elimination of a space on the right end of all the elements
§11.4.30	`strreplace()`	String search and replacement of all the elements
§11.4.31	`regreplace()`	String search and replacement of all the elements using a regular expression
§11.4.32	`tolower()`	Replaces the uppercase version of characters in all the elements with the lowercase version
§11.4.33	`toupper()`	Replaces the lowercase version of characters in all the elements with the uppercase version
§11.4.34	`expand_tabs()`	Replaces TAB characters in all the elements with a white space character
§11.4.35	`contract_spaces()`	Replaces white space characters in all the elements with a TAB character

Table 23: List of member functions available for use with the asarray_tstring class (Continued from previous page)

11.3 Operators

11.3.1 []

NAME

`[]` — Reference to the element value object (the tstring class) corresponding to a specified key

SYNOPSIS

```
tstring &operator[]( const char *key );  ..... 1
const tstring &operator[]( const char *key ) const;  ..... 2
```

DESCRIPTION

Returns a reference to the element value object (tstring class; §9) in an associative array corresponding to a key. "`[]`" can be immediately followed by "`.`" to connect to use of tstring class member functions (§9) (The EXAMPLE uses the tstring class "`=`" operator and assign() member function).

Member function 1 is for both reading and writing and operates in the same manner as the at() member function does. Member function 2 is for reading only and operates in the same manner as the at_cs() member function does.

A key string that does not exist being specified results in a set of the specified key string and the value `""` being added to the associative array with member function 1 while with member function 2 an exception occurs.

Whether member function 1 or the member function 2 is used is automatically determined by the presence or absence of the "const" attribute for an object. Member function 1 is automatically selected if the object does not have a "const" attribute but member function 2 is if it does.

For more details on `at()` and `at_cs()` refer to §11.4.4.

PARAMETER

[I] `key` Key string in an associative array

RETURN VALUE

Reference to the element value object (the tstring class) in an associative array corresponding to a key

EXCEPTION

If a specified key string is NULL.
If the system failed to secure an internal buffer (Member function 1).
If a key string that does not exist is specified (Member function 2).

EXAMPLE

The following code sets a key and a value to the associative array object `my_asarr`. The operator "`=`" and `assign()` operate in the same manner:

```
asarray_tstring my_asarr;
my_asarr["google"] = "Larry Page";
my_asarr["YouTube"].assign("Steve Chen");
```

11.3.2 =

NAME

= — Copies objects for the `asarray_tstring` class

SYNOPSIS

```
asarray_tstring &operator=(const asarray_tstring &obj);
```

DESCRIPTION

Assigns to itself the object for the `asarray_tstring` class specified on the right (argument) of the operator.

PARAMETER

[I] `obj` asarray_tstring class object

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory.

EXAMPLE

The following code assigns the associative array object `my_asarr` to the associative array object `my_asarrObj`, and then prints the result to standard output. For more details on `cstr()` refer to the descriptions provided in §11.4.3:

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["linux"]   = "Linus Torvalds";
my_asarr["windows"] = "Bill Gates";
my_asarr["mac"]     = "Steve Jobs";

asarray_tstring my_asarrObj;
my_asarrObj = my_asarr;
/* Display all the elements */
for ( size_t i=0 ; i < my_asarrObj.length() ; i++ ) {
    const char *key = my_asarrObj.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarrObj.cstr(key));
}
```

Result of execution

```
linux ...  [Linus Torvalds]
windows ...  [Bill Gates]
mac ...  [Steve Jobs]
```

11.4 Member functions

11.4.1 length()

NAME

length() — Length of an associative array (the number of arrays), and the length of a value string

SYNOPSIS

```
size_t length() const;  ..... 1
size_t length( const char *key ) const;  ..... 2
```

DESCRIPTION

Member function 1 returns the length of an associative array (number of arrays).

Member function 2 returns the string length of a value corresponding to the key string specified by an argument.

PARAMETER

[I] `key` Key string in an associative array
([I] : Input, [O] : Output)

RETURN VALUE

Number of elements of an associative array or the string length of a value corresponding to a specified key

EXAMPLE

The following code prints to standard output the length of arrays in the associative array `my_asarr`, and the string length of the value `"Larry Page"` corresponding to the key `"google"`:

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["linux"]   = "Linus Torvalds";
my_asarr["google"]  = "Larry Page";
my_asarr["mac"]     = "Steve Jobs";

sio.printf("my_asarr total length   ... [%zu]\n", my_asarr.length());
sio.printf("my_asarr key='google' length ... [%zu]\n",
            my_asarr.length("google"));
```

Result of execution

```
my_asarr total length ...  [3]
my_asarr key='google' length ...  [10]
```

11.4.2 cstrarray()

NAME

cstrarray() — Pointer array (NULL-terminated) for a value string in an associative array

SYNOPSIS

```
const char *const *cstrarray() const;
```

DESCRIPTION

Returns the pointer array for a value string in an associative array. Pointer arrays are always NULL-terminated.

RETURN VALUE

The pointer array (NULL-terminated) to a value string in an associative array

EXAMPLE

The following code acquires the pointer array for a value string in the associative array `my_asarr`, and then prints the value to standard output.

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["MIT"] = "cambridge";
my_asarr["Princeton"] = "New Jersey";
my_asarr["Berkeley"] = "California";

const char *const *my_ptr;
my_ptr = my_asarr.cstrarray();
if ( my_ptr != NULL ) {
    for ( int i = 0 ; my_ptr[i] != NULL ; i++ ) {
        sio.printf("%d ... [%s]\n", i, my_ptr[i]);
    }
}
```

Result of execution

```
0 ...  [cambridge]
1 ...  [New Jersey]
2 ...  [California]
```

11.4.3 cstr(), c_str(), cstrf(), vcstrf()

NAME

cstr(), c_str(), cstrf(), vcstrf() — Value string corresponding to a specified key or element number

SYNOPSIS

```
const char *cstr( const char *key ) const;  ..................................  1
const char *c_str( const char *key ) const;  ..................................  2
const char *cstrf( const char *fmt, ... ) const;  ..........................  3
const char *vcstrf( const char *fmt, va_list ap ) const;  ..................  4
const char *cstr( size_t index ) const;  ....................................  5
```

DESCRIPTION

Member functions 1 and 2 return the value string in an associative array corresponding to a specified key.

Member functions 3 and 4 enable the key string you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member function 3 converts each element of data of a variable-length argument depending on the conversion specifications set in `fmt`. Member function 4 converts the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

Member function 5 returns the value string in an associative array corresponding to a specified element number. Please note that the element number for the first element of arrays is always `0`.

If a key is invalid or an element number has a value larger than the length of an array specified to it `NULL` is returned to indicate the error.

PARAMETER

[I] `key` Key string in an associative array object
[I] `fmt` Format specifications for a key string
[I] `...` Each element of data of a variable-length argument supporting `fmt`
[I] `ap` List of variable-length arguments supporting `fmt`
[I] `index` Element number
([I] : Input, [O] : Output)

RETURN VALUE

The address for the string corresponding to a specified key or element number (Normal termination)
NULL (Error) : If a key or element number is invalid.

EXAMPLE

The following code prints to standard output the string corresponding to the key `"Riyuu"` for the associative array `my_asarr`:

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["Itamu Hito"]    = "Arata Tendou";
my_asarr["Kucyu Buranko"] = "Hideo Okuda";
my_asarr["Riyuu"]         = "Miyuki Miyabe";

sio.printf("my_asarr c_str ... [%s]\n", my_asarr.c_str("Riyuu"));
```

Result of execution

```
my_asarr c_str ...  [Miyuki Miyabe]
```

11.4.4 at(), atf()

NAME

at(), atf() — Reference to the element value object (the tstring class) corresponding to a specified key or element number

SYNOPSIS

```
tstring &at( const char *key );  ..... 1
tstring &atf( const char *fmt, ... );  ..... 2
tstring &vatf( const char *fmt, va_list ap );  ..... 3
tstring &at( size_t index );  ..... 4
const tstring &at( const char *key ) const;  ..... 5
const tstring &atf( const char *fmt, ... ) const;  ..... 6
const tstring &vatf( const char *fmt, va_list ap ) const;  ..... 7
const tstring &at( size_t index ) const;  ..... 8
```

DESCRIPTION

Returns a reference to the element value object (tstring class; §9)) corresponding to a key string (with member functions 1 to 3 and 5 to 7) or an element number (with member functions 4 and 8) in an argument. These member functions can be immediately followed by "`.`" to connect to use of tstring class member functions (§9) (The EXAMPLE uses the tstring class "`=`" operator and assign() member function). Member functions 1 to 4 can be

used both for reading and writing elements while member functions 5 to 8 are for reading only.

Member functions 2, 3, 6 and 7 enable a key string that you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member functions 2 and 6 convert each element of data of a variable-length argument depending on the conversion specifications set in `fmt`. Member functions 3 and 7 convert the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

If a key string that does not exist is specified, with member functions 1 to 3 a set of the specified key string and the value `""` is added to the associative array but with member functions 5 to 7 an exception occurs.

With member functions 4 and 8 if `index` has a value larger than the length of an array specified to it an exception occurs. Please note that the element number for the first element of arrays is always `0`.

Whether member functions 1 to 4 or member functions 5 to 8 are used is automatically determined by the presence or absence of the "const" attribute for an object. Member functions 1 to 4 are automatically selected if the object does not have the "const" attribute while member functions 5 to 8 are if it does.

PARAMETER

[I]	`key`	Key string in an associative array object
[I]	`fmt`	Format specifications for a key string
[I]	`...`	Each element of data of a variable-length argument supporting `fmt`
[I]	`ap`	List of variable-length arguments supporting `fmt`
[I]	`index`	Element number

([I] : Input, [O] : Output)

RETURN VALUE

Reference to the element value object (the tstring class) corresponding to a specified key or element number

EXCEPTION

If a specified key string is NULL.
If a key string that does not exist is specified (Member functions 5 to 7).
If a specified element number is invalid (Member functions 4 and 8).
If the system failed to secure an internal buffer (Member functions 1 to 3 and 6 and 7).

EXAMPLE

The following code adds the set of the key `"Yasushi Inoue"` and the value `"Tougyu"` to the associative array `my_asarr`, and then prints the result to standard output in order to verify it:

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["Takeshi kaikou"]  = "Hadaka no Oosama";
my_asarr["Koubou Abe"]      = "Kabe";
my_asarr["Kenzaburou Ooe"]  = "Shiiku";

my_asarr.at("Yasushi Inoue") = "Tougyu";
sio.printf("my_asarr c_str ... [%s]\n", my_asarr.at("Yasushi Inoue").cstr());
```

Result of execution

```
my_asarr c_str ...  [Tougyu]
```

11.4.5 at_cs(), atf_cs()

NAME

at_cs(), atf_cs() — Reference to the element value object (the tstring class) corresponding to a specified key or element number (Read only).

SYNOPSIS

```
const tstring &at_cs( const char *key ) const;  ..............................  1
const tstring &atf_cs( const char *fmt, ... ) const;  ........................  2
const tstring &vatf_cs( const char *fmt, va_list ap ) const;  ..............  3
const tstring &at_cs( size_t index ) const;  ..................................  4
```

DESCRIPTION

Returns a reference to the element value object (tstring class; §9) corresponding to a key. These member functions are for reading only.

Member functions 2 and 3 enable a key string that you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member function 2 converts each element of data of a variable-length argument depending on the conversion specifications set in `fmt`. Member function 3 converts the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

Please note that the element number for the first element of arrays is always 0.

If a key string or element number that does not exist is specified an exception occurs.

PARAMETER

[I]	`key`	Key string in an associative array object
[I]	`fmt`	Format specifications for a key string
[I]	`...`	Each element of data of a variable-length argument supporting `fmt`
[I]	`ap`	List of variable-length arguments supporting `fmt`
[I]	`index`	Element number

([I] : Input, [O] : Output)

RETURN VALUE

A reference to the element value object (tstring class) corresponding to a specified key or element number

EXCEPTION

If a specified key string is NULL.
If a key string that does not exist is specified (Member functions 1 to 3).
If a specified element number is invalid.
If the system failed to secure an internal buffer (Member functions 2 and 3).

11.4.6 index(), indexf(), vindexf()

NAME

index(), indexf(), vindexf() — Acquires the element number corresponding to a key string

SYNOPSIS

```
ssize_t index( const char *key ) const;  ..... 1
ssize_t indexf( const char *fmt, ... ) const;  ..... 2
size_t vindexf( const char *fmt, va_list ap ) const;  ..... 3
```

DESCRIPTION

Acquires the element number corresponding to a key string. Please note that the element number for the first element of arrays is always `0`.

Member functions 2 and 3 enable a key string that you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member function 2 converts each element of data of a variable-length argument depending on the conversion specifications set in `fmt`. Member function 3 converts the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

PARAMETER

[I] `key` Key string
[I] `fmt` Format specifications for a key string
[I] `...` Each element of data of a variable-length argument supporting `fmt`
[I] `ap` List of variable-length arguments supporting `fmt`
([I] : Input, [O] : Output)

RETURN VALUE

Non-negative value	:	If a specified key string is found the corresponding element number.
Negative value (Error)	:	If a specified key string is not found.

EXCEPTION

If the system failed to secure an internal buffer (Member functions 2 and 3).

EXAMPLE

The following code acquires the element number of which the key is `"Muritaniya"` in the associative array object `my_country`, and then prints the result to standard output:

```
stdstreamio sio;

asarray_tstring my_country;
my_country["Saudi Arabia"] = "Abdullah bin Abdulaziz al-Saud";
my_country["Muritaniya"] = "Mohamed Ould Abdel Aziz";
my_country["Cyprus"] = "Demetris Christofias";

sio.printf("my_country.index(\"Muritaniya\") ... [%zd]\n",
           my_country.index("Muritaniya"));
```

Result of execution

```
my_country.index("Muritaniya") ...  [1]
```

11.4.7 key()

NAME

key() — Acquires the key string corresponding to an element number

SYNOPSIS

```
const char *key( size_t index ) const;
```

DESCRIPTION

Acquires the key string corresponding to the element number specified by `index`. Please note that the element number for the first element of arrays is always `0`.

If `index` has a value larger than the length of an array specified to it `NULL` is returned.

PARAMETER

[I] `index` Element number
([I] : Input, [O] : Output)

RETURN VALUE

Address for the internal buffer for a key string (Normal termination)
NULL (Error) : If an element number that is larger than the length of an array is specified.

EXAMPLE

The following code acquires the keys for the associative array object `my_city` in the order of from `0`, and then prints the acquired keys and values to standard output:

```
stdstreamio sio;

asarray_tstring my_city;
my_city["Yokohama"] = "Fumiko Hayashi";
my_city["Osaka"]    = "Kunio Hiramatsu";
my_city["Fukuoka"]  = "Hiroshi Yoshida";

for ( size_t i=0 ; i < my_city.length() ; i++ ) {
    const char *key = my_city.key(i);
    sio.printf("#%zx -> %s ... [%s]\n", i, key, my_city.cstr(key));
}
```

Result of execution

```
#0 -> Yokohama ...  [Fumiko Hayashi]
#1 -> Osaka ...  [Kunio Hiramatsu]
#2 -> Fukuoka ...  [Hiroshi Yoshida]
```

11.4.8 keys()

NAME

keys() — References the array object for a key string (Read only)

SYNOPSIS

```
const tarray_tstring &keys() const;
```

DESCRIPTION

Returns a reference to the array object (tarray_tstring class; §10) for a key string managed inside an object. This member function can be immediately followed by "`.`" to connect to use of tarray_tstring class member functions. The tarray_tstring class member functions available for use are only those that do not change key strings, or that is, those that have the const attribute.

RETURN VALUE

Reference to the array object (tarray_tstring class) for a key string

---

11.4.9 values()

NAME

values() — References the array object for a value string (Read only)

SYNOPSIS

```
const tarray_tstring &values() const;
```

DESCRIPTION

Returns a reference to the array object (tarray_tstring class; §10)) for a value string managed inside an object. This member function can be immediately followed by "`.`" to connect to use of tarray_tstring class member functions. The tarray_tstring class member functions available for use are only those that do not change value strings, or that is, those that have the const attribute.

RETURN VALUE

Reference to the array object (the tarray_tstring class) for a key string

---

11.4.10 dprint()

NAME

dprint() — Outputs object information to standard error output (For user debugging)

SYNOPSIS

```
void dprint() const;
```

DESCRIPTION

Outputs information on an object to standard error output.

Member function designed for debugging user programs.

EXAMPLE

The following code outputs the information on the object `my_array` to standard error output. In the result of the execution the address for the object can be seen to be displayed within `[]`, but this does depend on the operating environment:

```
asarray_tstring my_array("CPU","Sparc", "OS","Solaris", NULL);
my_array.dprint();
```

Result of execution

```
sli::asarray_tstring[obj=0xbffff3d0] = { {"CPU", "Sparc"}, {"OS", "Solaris"} }
```

11.4.11 swap()

NAME

swap() — Interchange of objects

SYNOPSIS

```
asarray_tstring &swap( asarray_tstring &sobj );
```

DESCRIPTION

Interchanges the content of the associative array object `sobj` with the content of itself.

PARAMETER

[I/O] `sobj` asarray_tstring class object to be interchanged
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXAMPLE

The following code interchanges the elements of the associative array objects `my_africa` and `my_america`, and then prints the content of the respective objects to standard output in order to verify it:

```
stdstreamio sio;

asarray_tstring my_africa;
my_africa["Rwanda"] = "Kigali";
my_africa["Cameroon"] = "Yaounde";

asarray_tstring my_america;
my_america["Honduras"] = "Tegucigalpa";
my_america["Jamaica"] = "Kingston";

my_africa.swap(my_america);
for ( size_t i=0 ; i < my_africa.length() ; i++ ) {
    const char *africa_key = my_africa.key(i);
    const char *africa__key = my_america.key(i);
    sio.printf("[%s]:%s <===> [%s]:%s\n", africa_key,
                my_africa.cstr(africa_key),
                africa__key, my_africa.cstr(africa_key));
}
```

Result of execution

```
[Honduras]:Tegucigalpa <===> [Rwanda]:Tegucigalpa
[Jamaica]:Kingston <===> [Cameroon]:Kingston
```

11.4.12 init()

NAME

init() — Complete initialization of objects

SYNOPSIS

```
asarray_tstring &init();  ..... 1
asarray_tstring &init(const asarray_tstring &obj);  ..... 2
```

DESCRIPTION

Initializes associative arrays.

Member function 1 completely initializes associative array objects. The memory area allocated to the array buffer and string buffer etc inside an associative array object is entirely released. If the `cstrarray()` member function (§11.4.2) is executed after `init()` is executed `NULL` is returned.

Member function 2 initializes objects with the content of `obj` (copies all the content of `obj` to itself).

PARAMETER

[I] `obj` asarray_tstring class object
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.
If the system encountered any corrupt memory (Member function 2).

EXAMPLE

The following code initializes the associative array `my_asarr` with `IgNobel_asarr`, and then prints the result to standard output in order to verify it:

```
stdstreamio sio;

asarray_tstring IgNobel_asarr;
asarray_tstring my_asarr;
IgNobel_asarr["2008"] = "Toshiyuki Nakagaki";
IgNobel_asarr["2007"] = "Mayu Yamamoto";
IgNobel_asarr["2006"] = "Dr.Nakamatsu";

my_asarr.init(IgNobel_asarr);
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
2008 ...  [Toshiyuki Nakagaki]
2007 ...  [Mayu Yamamoto]
2006 ...  [Dr.Nakamatsu]
```

11.4.13 assign(), assignf(), vassignf()

NAME

assign(), assignf(), vassignf() — Initialization of objects and assignment of elements (Specifies a single set of a key and a value)

SYNOPSIS

```
asarray_tstring &assign( const char *key, const char *val );  ..............  1
asarray_tstring &assign( const char *key, const tstring &val );  ...........  2
asarray_tstring &assignf( const char *key, const char *fmt, ... );  .......  3
asarray_tstring &vassignf( const char *key, const char *fmt, va_list ap );  4
```

DESCRIPTION

Initializes associative array objects with a specified single element (combination of a key and a value).

Member functions 1 and 2 initialize objects with the key `key` and the value `val`.

Member functions 3 and 4 enable a value string that you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member function

3 converts each element of data of a variable-length argument depending on the conversion specifications set in `fmt`. Member function 4 converts the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

PARAMETER

[I] `key` Key string to be set to an associative array object
[I] `val` Value string to be set to an associative array object
[I] `fmt` Format specifications for a value string to be set
[I] `...` Each element of data of a variable-length argument supporting `fmt`
[I] `ap` List of variable-length arguments supporting `fmt`
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code sets a single set of a key and a value to the associative array `my_asarr`, and then prints the result to standard output:

```
stdstreamio sio;

asarray_tstring my_asarr;

const char *key0 = "Everest";
const char *val0 = "Nepal";
my_asarr.assign(key0,val0);
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
Everest ...  [Nepal]
```

11.4.14 assign(), vassign()

NAME

assign(), vassign() — Initialization of objects and assignment of elements (Specifies multiple sets of a key and a value)

SYNOPSIS

```
asarray_tstring &assign( const asarray_tstring &src );  ..... 1
asarray_tstring &assign( const asarrdef_tstring elements[] );  ..... 2
asarray_tstring &assign( const asarrdef_tstring elements[], size_t n );  . 3
asarray_tstring &assign( const char *key0, const char *val0,
                         const char *key1, ... );  ..... 4
asarray_tstring &vassign( const char *key0, const char *val0,
                          const char *key1, va_list ap );  ..... 5
```

DESCRIPTION

Initializes associative array objects with specified multiple elements (combinations of a key and a value).

Member function 1 assigns to itself the content of the asarray_tstring class object `src`.

Member functions 2 and 3 set the content of asarrdef_tstring type (structure) arrays (With member function 2 `elements` must be terminated at {NULL,NULL}). Member function 3 sets `n` elements from the beginning of `elements`. If `n` is larger than the number of `elements` (until reaching {NULL,NULL}) is specified `n` is ignored.

Member functions 4 and 5 create associative arrays using multiple combinations of a key of const char * type and a value specified in `key0`, `val0`, `key1` and the variable-length arguments thereafter (must be NULL-terminated).

PARAMETER

[I]	`src`	asarray_tstring class object that includes the element to be sourced
[I]	`elements`	Array of asarrdef_tstring type (structure) that includes the element to be sourced (With member function 2 must be terminated at {NULL,NULL})
[I]	`n`	Number of array `elements`
[I]	`key0`, `key1`	Key string to be set to an associative array
[I]	`val0`	Value string to be set to an associative array
[I]	`...`	Each element of data of a variable-length argument for the string that is a key/value (Needs to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for the string that is a key/value (Needs to be `NULL`-terminated)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code sets the content of a structure asarrdef_tstring to the associative array `my_asarr`, and then prints the result to standard output. At the end of the array the key and the value are both set to `NULL`:

```
stdstreamio sio;

asarray_tstring my_asarr;
const asarrdef_tstring mount_elem[] = { {"K2","China"},
        {"Kangchenjunga","Nepal"}, {"Mount Kenya","Kenya"}, {NULL,NULL} };

my_asarr.assign(mount_elem);
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
K2 ...  [China]
```

```
Kangchenjunga ...  [Nepal]
Mount Kenya ...  [Kenya]
```

11.4.15 assign_keys()

NAME

assign_keys() — Sets multiple strings or a string array to keys

SYNOPSIS

```
asarray_tstring &assign_keys( const char *key0, ... );  ..... 1
asarray_tstring &vassign_keys( const char *key0, va_list ap );  ..... 2
asarray_tstring &assign_keys( const char *const *keys );  ..... 3
asarray_tstring &assign_keys( const tarray_tstring &keys );  ..... 4
```

DESCRIPTION

Sets the specified multiple strings `key0, ...` or string array `keys` to keys for an associative array.

The number of keys specified in arguments becomes the number of elements in the associative array (Associative array elements of the number exceeding the number of keys specified by an argument are deleted).

Variable arguments for member functions 1 and 2 and the pointer array `keys` for member function 3 must be `NULL`-terminated.

PARAMETER

- [I] `key0` Key string
- [I] `...` Each element of data of a variable-length argument for a key string (`NULL`-terminated)
- [I] `ap` List of variable-length arguments for a key string (`NULL`-terminated)
- [I] `keys` String array to be set to a key string (With member function 3 must be `NULL`-terminated)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

Refer to the EXAMPLE in §11.4.16.

11.4.16 assign_values()

NAME

assign_values() — Sets multiple strings or a string array to values

SYNOPSIS

```
asarray_tstring &assign_values( const char *val0, ... );  ..... 1
asarray_tstring &vassign_values( const char *val0, va_list ap );  ..... 2
asarray_tstring &assign_values( const char *const *values );  ..... 3
asarray_tstring &assign_values( const tarray_tstring &values );  ..... 4
```

DESCRIPTION

Sets the specified multiple strings `val0, ...` or string array `values` to values for an associative array.

If the number of values specified by an argument exceeds the number for the associative array inside an object the values of the number that exceeds are discarded.

Variable arguments for member functions 1 and 2 and the pointer array `values` for member function 3 must be `NULL`-terminated.

PARAMETER

[I]	`val0`	Value string
[I]	`...`	Each element of data of a variable-length argument for a value string (`NULL`-terminated)
[I]	`ap`	List of variable-length arguments for a value string (`NULL`-terminated)
[I]	`values`	String array to be set to a value string (With member function 3 must be `NULL`-terminated)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

Initializes the associative array using `assign_keys()` and `assign_values()`:

```
asarray_tstring my_array;
my_array.assign_keys("CPU", "OS", NULL);
my_array.assign_values("PentiumPro", "Linux", NULL);
my_array.dprint();
```

Result of execution

```
sli::asarray_tstring[obj=0xbffff3d0] = { {"CPU", "PentiumPro"}, {"OS", "Linux"} }
```

11.4.17 split_keys()

NAME

split_keys() — Divides strings and sets them to keys

SYNOPSIS

```
asarray_tstring &split_keys( const char *src_str, const char *delims,
                             bool zero_str, const char *quotations,
                             int escape, bool rm_escape );  ..... 1
asarray_tstring &split_keys( const char *src_str, const char *delims,
                             bool zero_str = false );  ..... 2
asarray_tstring &split_keys( const tstring &src_str, const char *delims,
                             bool zero_str, const char *quotations,
                             int escape, bool rm_escape );  ..... 3
asarray_tstring &split_keys( const tstring &src_str, const char *delims,
                             bool zero_str = false );  ..... 4
```

DESCRIPTION

Divides the string `src_str` with the delimiter characters and then sets them to the keys for an associative array. The delimiter characters are given by character set of `delims` argument, and `delims` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

The number of keys obtained after dividing the string becomes the number of elements in the associative array (Associative array elements of the number exceeding the number acquired after dividing the string are deleted).

`zero_str` can be specified to indicate whether to allow the string length of zero for key elements after the division. If `zero_str` is `false` key elements with the string length of `0` cannot be created. If `zero_str` is `true` key elements with the string length of `0` can be created (used with csv format etc). If `zero_str` is not specified it is treated as `false`.

If you do not want to divide strings that are parenthesized with "specific characters" such as quotation marks etc., such "specific characters" can be specified using `quotations`. For example, if you want to exclude strings parenthesized by a single quotation from the strings to be divided specify `"'"`.

An escape character can be specified using `escape`. If you want to delete escape any characters that come after the division set `rm_escape` to `true`. However, any escape characters in the strings that are parenthesized by a character specified by `quotations` will not be deleted.

If you cannot successfully retrieve keys using only this member function a method of first creating a key string in a tarray_tstring class also exists, following which a key can then be set using the assign_keys() member function (§11.4.15).

PARAMETER

[I]	`src_str`	String to be divided
[I]	`delims`	String that includes delimiter characters
[I]	`zero_str`	Whether or not to allow strings with the length of 0 as a result of delimiting (true/false)
[I]	`quotations`	String that includes quotation characters
[I]	`escape`	Escape character
[I]	`rm_escape`	Flag to indicate whether or not to delete escape characters (true/false)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

Refer to the `EXAMPLE` in §11.4.18.

An example of using member function 2 is provided in §3.5.5.

11.4.18 split_values()

NAME

split_values() — Divides strings and sets them to values

SYNOPSIS

```
asarray_tstring &split_values( const char *src_str, const char *delims,
                               bool zero_str, const char *quotations,
                               int escape, bool rm_escape );  ..... 1
asarray_tstring &split_values( const char *src_str, const char *delims,
                               bool zero_str = false );  ..... 2
asarray_tstring &split_values( const tstring &src_str, const char *delims,
                               bool zero_str, const char *quotations,
                               int escape, bool rm_escape );  ..... 3
asarray_tstring &split_values( const tstring &src_str, const char *delims,
                               bool zero_str = false );  ..... 4
```

DESCRIPTION

Divides the string `src_str` with the delimiter characters, and then sets them to the values for an associative array. The delimiter characters are given by character set of `delims` argument, and `delims` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

If the number of values acquired after dividing the string exceeds the number for the associative array inside an object the value of the exceeding number is discarded.

`zero_str` can be used to specify whether to allow a string length of zero for value elements after the division. If `zero_str` is `false` value elements with a string length of `0` cannot be created. If `zero_str` is `true` value elements with a string length of `0` can be created (used with the csv format etc). If `zero_str` is not specified it is treated as `false`.

If you do not want to divide strings that are parenthesized by a "specific character" such as a quotation etc a "specific character" can be specified using `quotations`. For example, if you want to exclude strings parenthesized by a single quotation from the strings to be divided specify `"'"`.

Escape characters are specified by `escape`. If you want to delete any escape characters remaining after the division set `rm_escape` to `true`. However, any escape characters in strings parenthesized by a character specified in `quotations` cannot be deleted.

PARAMETER

[I]	`src_str`	String to be divided
[I]	`delims`	String that includes delimiter characters
[I]	`zero_str`	Whether or not to allow strings with a length of 0 as a result of delimiting (true/false)
[I]	`quotations`	String that includes a quotation character
[I]	`escape`	Escape character
[I]	`rm_escape`	Flag that indicates whether or not to delete escape characters (true/false)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code specifies a delimiter character " " and a quotation character, divides a string, and then sets the divided string as keys to the associative array `my_arr`. The delimiter character "," and a quotation character are then specified, and a string divided, and divided string set as values to the associative array `my_arr`. It then standard-outputs the set keys and values to verify them:

```
stdstreamio sio;

const char *line =
      "'Camellia sasanqua' 'Chrysanthemum morifolium' 'Cyclamen persicum'";
asarray_tstring my_arr;
my_arr.split_keys(line, " ", false, "'", 0);
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    const char *key = my_arr.key(i);
    sio.printf("%s ... [%s]\n", key, my_arr.cstr(key));
}

const char *val =
           "'Camellia,pink','Chrysanthemum,yellow','Cyclamen,pink'";
my_arr.split_values(val, ",", false, "'", 0);
for ( size_t i=0 ; i < my_arr.length() ; i++ ) {
    const char *key = my_arr.key(i);
    sio.printf("%s ... [%s]\n", key, my_arr.cstr(key));
}
```

Result of execution

```
'Camellia sasanqua' ...  ['Camellia,pink']
'Chrysanthemum morifolium' ...  ['Chrysanthemum,yellow']
'Cyclamen persicum' ...  ['Cyclamen,pink']
```

Examples of using member function 2 is provided in §3.5.5 and §11.4.27.

11.4.19 append(), appendf(), vappendf()

NAME

append(), appendf(), vappendf() — Adds elements (Specifies a single set of a key and a value)

SYNOPSIS

```
asarray_tstring &append( const char *key, const char *val );  ..............  1
asarray_tstring &append( const char *key, const tstring &val );  ...........  2
asarray_tstring &appendf( const char *key, const char *fmt, ... );  .......  3
asarray_tstring &vappendf( const char *key, const char *fmt, va_list ap );  4
```

DESCRIPTION

Adds a specified single element (combination of a key and a value) to an associative array object.

Member functions 1 and 2 add the key `key` and the value `val`.

Member functions 3 and 4 enable a value string that you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member function 3 converts each element of data of a variable-length argument depending on the conversion

specifications set in `fmt`. Member function 4 converts the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

If duplicate keys exist a warning is output to standard error output at the time of execution, and the function not processed.

PARAMETER

[I] `key` Key string to be added to an associative array object
[I] `val` Value string to be added to an associative array object
[I] `fmt` Format specifications for a value string to be added
[I] `...` Each element of data of a variable-length argument supporting `fmt`
[I] `ap` List of variable-length arguments supporting `fmt`
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code adds a single element to the associative array object `my_asarr` using the appendf() member function, and then prints the result to standard output:

```
stdstreamio sio;

asarray_tstring my_asarr("rice","China", "coffee","Brazil", NULL);
const char *key_cacao = "cacao";

my_asarr.appendf(key_cacao,"### No.1 is %s ###","Cote d'Ivoire");
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
rice ...  [China]
coffee ...  [Brazil]
cacao ...  [### No.1 is Cote d'Ivoire ###]
```

11.4.20 append(), vappend()

NAME

append(), vappend() — Adds elements (Specifies multiple sets of a key and a value)

SYNOPSIS

```
asarray_tstring &append( const asarray_tstring &src );  ..... 1
asarray_tstring &append( const asarrdef_tstring elements[] );  ..... 2
asarray_tstring &append( const asarrdef_tstring elements[], size_t n );  . 3
asarray_tstring &append( const char *key0, const char *val0,
                         const char *key1, ... );  ..... 4
asarray_tstring &vappend( const char *key0, const char *val0,
                          const char *key1, va_list ap );  ..... 5
```

DESCRIPTION

Adds specified multiple elements (combinations of a key and a value) to an associative array object.

Member function 1 adds the content of the asarray_tstring class object `src`.

Member functions 2 and 3 add the content of asarrdef_tstring type (structure) arrays (With member function 2 `elements` must be terminated at {NULL,NULL}). Member function 3 adds `n` elements from the beginning of `elements`. If `n` larger than the number of `elements` (until reaching ({NULL,NULL}) is specified `n` is ignored.

Member functions 4 and 5 add multiple combinations of a key of const char * type and a value specified in `key0`, `val0`, `key1` and the variable-length arguments thereafter as associative array elements (must be NULL-terminated).

If duplicate keys exist a warning is output to standard error output at the time of execution, and the function not processed.

PARAMETER

[I]	`src`	asarray_tstring class object that includes the element to be sourced
[I]	`elements`	Array of asarrdef_tstring type (structure) that includes the element to be sourced (With member function 2 must be terminated at {NULL,NULL})
[I]	`n`	Number of the array `elements`
[I]	`key0`, `key1`	Key string to be added to an associative array object
[I]	`val0`	Value string to be added to an associative array object
[I]	`...`	Each element of data of a variable-length argument for the string that is a key/value (Needs to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for the string that is a key/value (Needs to be `NULL`-terminated)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code adds to the associative array object `my_asarr` the string array `foods` that is defined using combinations of a key and a value, and then prints the result to standard output. `NULL` is set both to the key and the value at the end of the array `foods`:

```
stdstreamio sio;

asarray_tstring my_asarr("rice","China", "coffee","Brazil", NULL);
const asarrdef_tstring foods[] = { {"banana","India"},
                                   {"wheat","China"}, {NULL,NULL} };

my_asarr.append(foods);
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
rice ...  [China]
coffee ...  [Brazil]
banana ...  [India]
wheat ...  [China]
```

11.4.21 insert(), insertf(), vinsertf()

NAME

insert(), insertf(), vinsertf() — Inserts elements (Specifies a single set of a key and a value)

SYNOPSIS

```
asarray_tstring &insert( const char *key,
                         const char *newkey, const char *newval ); ........ 1
asarray_tstring &insert( const char *key,
                         const char *newkey, const tstring &newval ); .... 2
asarray_tstring &insertf( const char *key,
                          const char *newkey, const char *fmt, ... ); .... 3
asarray_tstring &vinsertf( const char *key,
                           const char *newkey, const char *fmt, va_list ap ); 4
```

DESCRIPTION

Inserts a specified single element (combination of a key and a value) before the element position of the key `key` in an associative array object.

Member functions 1 and 2 insert the element of a combination of the key `newkey` and the value `newval`.

Member functions 3 and 4 enable a value string that you want to specify to be set in the same format and with the same variable arguments as the printf() function. Member function 3 converts each element of data of a variable-length argument depending on the conversion specifications set in `fmt`. Member function 4 converts the list `ap` of variable-length arguments depending on the conversion specifications set in `fmt`. For more details on `fmt` refer to the descriptions provided in §8.1.14.

If duplicate keys exist a warning is output to standard error output at the time of execution, and the function not processed.

PARAMETER

[I]	`key`	String for a key for an associative array object that is in the insert position (Inserted before the key)
[I]	`newkey`	Key string to be inserted in an associative array object
[I]	`newval`	Value string to be inserted in an associative array object
[I]	`fmt`	Format specifications for a value string to be inserted
[I]	`...`	Each element of data of a variable-length argument supporting `fmt`
[I]	`ap`	List of variable-length arguments supporting `fmt`

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code adds a single element to the associative array object `my_asarr`, and then prints the result to standard output:

```
stdstreamio sio;

asarray_tstring my_asarr("Nile River","Africa", "the Amazon","South America",
                         NULL);
my_asarr.insert("the Amazon", "Chang River", "China");
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
Nile River ...  [Africa]
Chang River ...  [China]
the Amazon ...  [South America]
```

11.4.22 insert(), vinsert()

NAME

insert(), vinsert() — Inserts elements (Specifies multiple sets of a key and a value)

SYNOPSIS

```
asarray_tstring &insert( const char *key,
                         const asarray_tstring &src );  ..................... 1
asarray_tstring &insert( const char *key,
                         const asarrdef_tstring elements[] );  ............. 2
asarray_tstring &insert( const char *key,
                         const asarrdef_tstring elements[], size_t n );  . 3
asarray_tstring &insert( const char *key,
                         const char *key0, const char *val0,
                         const char *key1, ... );  ........................... 4
asarray_tstring &vinsert( const char *key,
                          const char *key0, const char *val0,
                          const char *key1, va_list ap );  .................. 5
```

DESCRIPTION

Inserts specified multiple elements (combinations of a key and a value) before the element position of the key `key` in an associative array object.

Member function 1 inserts the content of the asarray_tstring class object `src`.

Member functions 2 and 3 insert the content of asarrdef_tstring type (structure) arrays (With member function 2 `elements` must be terminated at {NULL,NULL}). Member function 3 inserts `n` elements from the beginning of `elements`. If `n` larger than the number of `elements` (until reaching {NULL,NULL}) is specified `n` is ignored.

Member functions 4 and 5 insert multiple combinations of a key of const char * type and a value specified in `key0`, `val0`, `key1` and the variable-length arguments thereafter as associative array elements (must be NULL-terminated).

If duplicate keys exist a warning is output to standard error output at the time of execution, and the function not processed.

PARAMETER

[I]	`key`	Key string for an associative array object that is in an insert position (Inserted before the key)
[I]	`src`	asarray_tstring class object that includes the element to be sourced
[I]	`elements`	Array of asarrdef_tstring type (structure) that includes the element to be sourced (With member function 2 must be terminated at {NULL,NULL})
[I]	`n`	Number of the array `elements`
[I]	`key0`, `key1`	Key string to be inserted in an associative array object
[I]	`val0`	Value string to be inserted in an associative array object
[I]	`...`	Each element of data of a variable-length argument for the string that is a key/value (Needs to be `NULL`-terminated)
[I]	`ap`	List of variable-length arguments for the string that is a key/value (Needs to be `NULL`-terminated)

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code inserts the associative array object `my_lake` in the associative array object `my_asarr`, and then prints the result to standard output. `NULL` is set both to the key and the value at the end of the array `lakes`:

```
stdstreamio sio;

const asarrdef_tstring lakes[] = { {"Lake Superior","North America"},
                                   {"Lake Victoria","Tanzania"}, {NULL,NULL} };
asarray_tstring my_lake(lakes);
asarray_tstring my_asarr("Caspian Sea","Eurasia",
                         "Aral Sea","Kazakhstan", NULL);

my_asarr.insert("Aral Sea", my_lake);
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
Caspian Sea ...  [Eurasia]
Lake Superior ...  [North America]
Lake Victoria ...  [Tanzania]
Aral Sea ...  [Kazakhstan]
```

11.4.23 erase()

NAME

erase() — Deletion of elements (sets of a key and a value)

SYNOPSIS

```
asarray_tstring &erase();  ..... 1
asarray_tstring &erase( const char *key, size_t num_elements = 1 );  ..... 2
```

DESCRIPTION

Deletes elements of an associative array object.

Member function 1 deletes all the elements (Array length becomes zero).

Member function 2 deletes `num_elements` elements starting from the element corresponding to the key specified by `key`. If `num_elements` is not specified an element is deleted.

The array length becomes smaller by the same length as deleted.

PARAMETER

[I]	`key`	Key string
[I]	`num_elements`	Number of elements to be deleted

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code deletes the key `"Democratic People's Republic of Korea"` and the value for the associative array object `my_asarr`, and then prints to standard output the content of `my_asarr` that remains after the deletion:

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["Kingdom of Lesotho"] = "Letsie III";
my_asarr["Democratic People's Republic of Korea"] = "Kim Jong-il";
my_asarr["Republic of Cote d'Ivoire"] = "Laurent Gbagbo";

my_asarr.erase("Democratic People's Republic of Korea");
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
Kingdom of Lesotho ...  [Letsie III]
Republic of Cote d'Ivoire ...  [Laurent Gbagbo]
```

11.4.24 clean()

NAME

clean() — Pads all the element values of an existing associative array with any string

SYNOPSIS

```
asarray_tstring &clean(const char *str = "");  ..... 1
asarray_tstring &clean(const tstring &str);  ..... 2
```

DESCRIPTION

Pads all the element values of an associative array with the character string `str`. The function can be used without specifying `str` but will be then processed as though a string of length 0 had been specified. Executing `clean()` does not change the key or the array length.

PARAMETER

[I] `str` String to pad a value in an associative array with
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code pads each of the strings that the associative array `my_asarr` includes with `Grammy Award`, and then prints the result to standard output:

```
stdstreamio sio;

asarray_tstring my_asarr;
my_asarr["U2"] = "Beautiful Day";
my_asarr["Eric Clapton"] = "Tears In Heaven";
my_asarr["TOTO"] = "Rosanna";

my_asarr.clean("Grammy Award");
for ( size_t i=0 ; i < my_asarr.length() ; i++ ) {
    const char *key = my_asarr.key(i);
    sio.printf("%s ... [%s]\n", key, my_asarr.cstr(key));
}
```

Result of execution

```
U2 ...  [Grammy Award]
Eric Clapton ...  [Grammy Award]
TOTO ...  [Grammy Award]
```

11.4.25 rename_a_key()

NAME

rename_a_key() — Changes key strings

SYNOPSIS

```
asarray_tstring &rename_a_key( const char *org_key, const char *new_key );
```

DESCRIPTION

Changes the key string `org_key` to the string specified by `new_key`.

If a key string that does not exist in an object is specified to `org_key` or `new_key` is a duplicate key string an error message output to standard error output.

PARAMETER

[I] `org_key` Original key string
[I] `new_key` Key string after being changed
([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXCEPTION
If the system failed to secure an internal buffer.

11.4.26 chomp()

NAME
chomp() — Elimination of newline characters in value strings of all the elements

SYNOPSIS
```
asarray_tstring &chomp( const char *rs = "\n" );
asarray_tstring &chomp( const tstring &rs );
```

DESCRIPTION
Eliminates a newline character on the right end of value strings in all the elements of a string associative array.

This member function executes the tstring class chomp() member function (§9.5.25) on all the elements of an associative array. For more details refer to §9.5.25.

PARAMETER
[I] `rs` Newline character string
([I] : Input, [O] : Output)

RETURN VALUE
Reference to itself

EXCEPTION
If the system failed to secure an internal buffer.

11.4.27 trim()

NAME
trim() — Elimination of spaces at both ends of value strings of all the elements

SYNOPSIS
```
asarray_tstring &trim( const char *side_spaces = " \t\n\r\f\v" );
asarray_tstring &trim( const tstring &side_spaces );
asarray_tstring &trim( int side_space );
```

DESCRIPTION
Eliminates arbitrary characters on both ends of value strings in all the elements of a string associative array.

`side_spaces` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

This member function executes the tstring class trim() member function on all the elements of an associative array. For more details refer to §9.5.26.

PARAMETER

[I] `side_space` Arbitrary character
[I] `side_spaces` Arbitrary character string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code divides a CSV-format string into elements using the split_values() member function (§11.4.18)), assigns them each as a value for an associative array object, and then eliminates any unnecessary white space characters on the right and left ends of each element using `trim()`:

```
stdstreamio sio;
asarray_tstring my_arr;
size_t i;
my_arr.assign_keys("CPU", "ChipSet", NULL);
my_arr.split_values(" Pentium4, E7205   ", ",", true);
for ( i=0 ; i < my_arr.length() ; i++ ) {
    const char *key = my_arr.key(i);
    sio.printf("%s ... [%s]\n", key, my_arr.cstr(key));
}
my_arr.trim();
for ( i=0 ; i < my_arr.length() ; i++ ) {
    const char *key = my_arr.key(i);
    sio.printf("%s ... [%s]\n", key, my_arr.cstr(key));
}
```

Result of execution

```
CPU ... [ Pentium4]
ChipSet ... [ E7205   ]
CPU ... [Pentium4]
ChipSet ... [E7205]
```

11.4.28 ltrim()

NAME

ltrim() — Elimination of a space on the left end of value strings of all the elements

SYNOPSIS

```
asarray_tstring &ltrim( const char *side_spaces = " \t\n\r\f\v" );
asarray_tstring &ltrim( const tstring &side_spaces );
asarray_tstring &ltrim( int side_space );
```

DESCRIPTION

Eliminates an arbitrary character on the left end of value strings in all the elements of a string associative array.

`side_spaces` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

This member function executes the tstring class ltrim() member function on all the elements of an associative array. For more details refer to §9.5.27.

PARAMETER

[I] `side_space` Arbitrary character
[I] `side_spaces` Arbitrary character string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

11.4.29 rtrim()

NAME

rtrim() — Elimination of a space on the right end of value strings of all the elements

SYNOPSIS

```
asarray_tstring &rtrim( const char *side_spaces = " \t\n\r\f\v" );
asarray_tstring &rtrim( const tstring &side_spaces );
asarray_tstring &rtrim( int side_space );
```

DESCRIPTION

Eliminates an arbitrary character on the right end of value strings in all the elements of a string associative array.

`side_spaces` can be specified as a simple list of characters like `" \t"` as well as expressions like `"[A-Z]"` or `"[^A-Z]"` as in regular expressions. In addition, a character class can also be specified inside `"[...]"`. For the character classes that can be specified refer to the descriptions and Table 19 provided in §9.5.26.

This member function executes the tstring class rtrim() member function on all the elements of an associative array. For more details refer to §9.5.28.

PARAMETER

[I] `side_space` Arbitrary character
[I] `side_spaces` Arbitrary character string
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

11.4.30 strreplace()

NAME

strreplace() — String search and replacement of value strings in all the elements

SYNOPSIS

```
asarray_tstring &strreplace( const char *org_str, const char *new_str,
                             bool all = false );
asarray_tstring &strreplace( const tstring &org_str, const char *new_str,
                             bool all = false );
asarray_tstring &strreplace( const char *org_str, const tstring &new_str,
                             bool all = false );
asarray_tstring &strreplace( const tstring &org_str, const tstring &new_str,
                             bool all = false );
```

DESCRIPTION

Searches for value strings in all the elements of a string associative array from the left side of a string for the string `org_str`, and if the string is found replaces it with the string `new_str`.

This member function executes the tstring class strreplace() member function on all the elements of an associative array (`0` is set to `pos`). For more details refer to Section §9.5.29.

PARAMETER

[I]	`org_str`	String to be detected
[I]	`new_str`	String to be sourced for replacement
[I]	`all`	Replace All flag

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code replaces the white space character in all the elements with the underscore "_":

```
stdstreamio sio;
asarray_tstring my_arr("OS","Solaris 9",
                       "VENDOR","Sun Microsystems, Inc.", NULL);
size_t i;
my_arr.strreplace(" ", "_", true);
for ( i=0 ; i < my_arr.length() ; i++ ) {
    const char *key = my_arr.key(i);
    sio.printf("%s ... [%s]\n", key, my_arr.cstr(key));
}
```

Result of execution

```
OS ... [Solaris_9]
VENDOR ... [Sun_Microsystems,_Inc.]
```

11.4.31 regreplace()

NAME

regreplace() — String search and replacement of value strings in all the elements using a regular expression

SYNOPSIS

```
asarray_tstring &regreplace( const char *pat,
                             const char *new_str, bool all = false );
asarray_tstring &regreplace( const tstring &pat,
                             const char *new_str, bool all = false );
asarray_tstring &regreplace( const tregex &pat,
                             const char *new_str, bool all = false );
asarray_tstring &regreplace( const char *pat,
                             const tstring &new_str, bool all = false );
asarray_tstring &regreplace( const tstring &pat,
                             const tstring &new_str, bool all = false );
asarray_tstring &regreplace( const tregex &pat,
                             const tstring &new_str, bool all = false );
```

DESCRIPTION

Replaces with the string `new_str` parts in value strings in all the elements of a string associative array that match the POSIX Extended Regular Expression (hereinafter referred to as a regular expression) specified by `pat`. The back references `"\\0"` through `"\\9"` can be used for `new_str` (`"\\0"` refers to the entire matching part). If you want to use the backslash itself specify `"\\\\"`.

This member function executes the tstring class regreplace() member function on all the elements of an associative array (`0` is set to `pos`). For more details refer to §9.5.30.

If you do not need a regular expression the strreplace() member function can be used (§11.4.30), which operates faster.

PARAMETER

[I]	`pat`	Character pattern (regular expression) or compiled object for the tregex class
[I]	`new_str`	String after the replacement
[I]	`all`	Replace All flag

([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

EXAMPLE

The following code deletes the escape character "\" from all the element values, attempts matching of the values of all the elements using the regular expression `"([\\])(.)"`, and if a string matches the expression replaces it with the back reference element 2:

```
stdstreamio sio;
asarray_tstring my_arr("OS","Solaris",
                       "VENDOR","Sun\\ Microsystems\\,\\ Inc.", NULL);
```

```
size_t i;
my_arr.regreplace("([\\])(.)", "\\2", true);
for ( i=0 ; i < my_arr.length() ; i++ ) {
    const char *key = my_arr.key(i);
    sio.printf("%s ... [%s]\n", key, my_arr.cstr(key));
}
```

Result of execution

```
OS ... [Solaris]
VENDOR ... [Sun Microsystems, Inc.]
```

11.4.32 tolower()

NAME

tolower() — Replaces the uppercase version of characters in value strings in all the elements with the lowercase version

SYNOPSIS

```
asarray_tstring &tolower();
```

DESCRIPTION

Replaces the uppercase version of alphabetical characters in value strings in all the elements of a string associative array with the lowercase version.

This member function executes the tstring class tolower() member function on all the elements of an associative array. For more details §9.5.31.

RETURN VALUE

Reference to itself

11.4.33 toupper()

NAME

toupper() — Replaces the lowercase version of characters in value strings in all the elements with the uppercase version

SYNOPSIS

```
asarray_tstring &toupper();
```

DESCRIPTION

Replaces the lowercase version of alphabetical characters in value strings in all the elements of a string associative array with the uppercase version.

This member function executes the tstring class toupper() member function on all the elements of an associative array. For more details refer to S 9.5.32.

RETURN VALUE

Reference to itself

11.4.34 expand_tabs()

NAME

expand_tabs() — Replaces TAB characters in value strings in all the elements with white space characters

SYNOPSIS

```
asarray_tstring &expand_tabs( size_t tab_width = 8 );
```

DESCRIPTION

Replaces horizontal tabulation characters `'\t'` in value strings in all the elements of a string associative array with a white space character, and then tabulates the characters to the value of `tab_width`.

This member function executes the tstring class expand_tabs() member function on all the elements of an associative array. For more details refer to §9.5.33.

PARAMETER

[I] `tab_width` TAB width
([I] : Input, [O] : Output)

RETURN VALUE

Reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

11.4.35 contract_spaces()

NAME

contract_spaces() — Replaces white space characters in value strings in all the elements with TAB characters

SYNOPSIS

```
asarray_tstring &contract_spaces( size_t tab_width = 8 );
```

DESCRIPTION

Replaces with `'\t'` all occurrences of two or more contiguous white space characters `' '` in value strings in all the elements of a string array that tabulate to the specified TAB width of `tab_width`.

This member function executes the tstring class contract_spaces() member function on all the elements of an associative array. For details more refer to §9.5.34.

PARAMETER

[I] `tab_width` TAB width
([I] : Input, [O] : Output)

RETURN VALUE

A reference to itself

EXCEPTION

If the system failed to secure an internal buffer.

---

12 MDARRAY_* Class

The mdarray_* class can handle multidimensional arrays of major types in C. It is possible to compute on multidimensional array in a manner similar to IDL.

It has features as follows:

- It has two operating modes: (1)"Automatic Resize Mode", in which the memory area is allocated automatically as necessary when users access an element of arrays and (2)"Manual Resize Mode", which is suitable for handling an image data.
- Classes (mdarray_float, mdarray_double, mdarray_int, etc.) corresponding to major numeric types (float, double, int, etc.) in C are prepared. It is possible to carry out an operation and assign values between different classes.
- Operators "`+`", "`-`", "`*`", "`/`", "`+=`", "`-=`", "`*=`", "`/=`", and "`=`" are available for scalar values and the whole array.
- Mathematic functions (`sin()`, `log()`, etc.) to carry out an operation to the whole array are prepared. (Almost all functions defined in math.h of libc are available.)

A list of available classes are shown in Table 24. Every class inherits the parent class "mdarray". (See the Advanced Version for the mdarray class.) By this implementation, operations on the whole array by operators "+", "-", "*", "/", "+=", "-=", "*=", "/=", and "=" are possible except some classes (sated as "N/A" for the Operation by Operators box in Table 24).

Class Name	Data Type in C	Operation by Operators
mdarray_float	float	OK
mdarray_double	double	OK
mdarray_uchar	unsigned char	OK
mdarray_short	short	OK
mdarray_int	int	OK
mdarray_long	long	OK
mdarray_llong	long long	OK
mdarray_int16	int16_t	OK
mdarray_int32	int32_t	OK
mdarray_int64	int64_t	OK
mdarray_size	size_t	N/A
mdarray_ssize	ssize_t	OK
mdarray_bool	bool	N/A
mdarray_uintptr	uintptr_t	N/A
mdarray_fcomplex	float complex	OK
mdarray_dcomplex	double complex	OK

Table 24: List of Available Classes

To use the classes shown in Table 24, write the following code followed by the user code:

```
#include <sli/mdarray.h>
#include <sli/mdarray_math.h>
```

`mdarray_math.h` is not necessary when only using classes, but it is necessary when using mathematic functions on the whole array. Additionally, write `using namespace sli;` in the code when namespace declaration(§4.1) is required.

A brief example of use is shown as follows:

```
#include <sli/stdstreamio.h>
#include <sli/mdarray.h>
#include <sli/mdarray_math.h>
using namespace sli;

int main()
{
    stdstreamio sio;
    size_t i;
    mdarray_long my_larr;
    mdarray_double my_darr;
    my_larr[0] = 100;
    my_larr[1] = 10;
    my_larr[2] = 1;
    my_darr = log10(my_larr);
    for ( i=0 ; i < my_darr.length() ; i++ ) {  /* print all elements of my_darr */
        sio.printf("%zu ... [%g]\n", i, my_darr[i]);
    }
}
```

Output:

```
0 ... [2]
1 ... [1]
2 ... [0]
```

12.1 How to Create an Object

An object can be initialized in several ways. At this time, the operating mode can be specified.

You can choose the Automatic Resize Mode or Manual Resize Mode.

In the Automatic Resize Mode, the number of dimensions and size of the array are extended automatically as necessary when you access an element of the array by using the operator `[]` or the member function `at()`, etc., or carry out an operation on the array by using the operators `+=`, `-=`, etc. On the other hand, in the Manual Resize Mode, the buffer size is not changed unless expressly specified to be resized. (This mode is suitable for handling an image data.)

As for the member functions whose behavior is different depending on the operating mode, "◯" is given in the Support for Operation Mode box in Table 26.

12.1.1 Method in which any Arguments are not Specified

In this case, the object is initialized in the Automatic Resize Mode.

```
    mdarray_double my_darr;
```

At this moment, any buffers are not allocated. After this, write the following code:

```
my_darr[3] = 1.23;
my_darr(3,2,1) = 4.56;
```

Buffers will be allocated automatically. The former case creates one dimensional array where the number of buffers is 4. The latter case creates three dimensional array of where the number of buffers is 4×3×2. The default value of an element of the array to which no value is given is 0. This default value can be changed by the assign_default() member function.

12.1.2 Method in which the Size of the Array is Specified

A size of the array can be specified when creating an object as follows:

```
mdarray_double my_darr(false, 1920,1080,3);
```

The operating mode has to be specified by the first argument. Give it `true` to initialize in the Automatic Resize Mode, otherwise give `false`. Specify the number of the elements of the first, second and third dimensions (up to third dimension) by the second argument. In the above example, an array size of 1920×1080×3 for the first, second and third dimensions, respectively, is allocated.

To allocate the n dimensional array, give the argument as follows:

```
const size_t nelemx = {800, 600, 3, 2}
mdarray_double my_darr(false, nelemx, 4);
```

This example shows the case of four dimensions. An array size of 800×600×3×2 for the first, second, third and fourth dimensions, respectively, is allocated.

12.1.3 Method in which the Size of the Array and the Default Value are Specified

There are two ways to give the default value to an object upon creation. One is to give the beginning address of the arbitrary buffer, and the other is to give the pointer array for the arbitrary buffer. Note that the number of dimensions is limited to 3 when the default value is given.

As shown below, **the operating mode have to be given to the first argument of the constructor**, followed by the number of elements of each dimension and the address of the array:

```
const double my_data1[] = {0.02, 0.2, 2.0};
mdarray_double my_darr1(false, 3, my_data1);
```

```
const double my_data2[][3] = {{0.02, 0.2, 2.0}, {20.0, 200.0, 2000.0}};
mdarray_double my_darr2(false, 3,2, *my_data2);
```

```
const double my_data3[][2][3] = {{{0.02, 0.2, 2.0}, {20.0, 200.0, 2000.0}},
                                 {{0.04, 0.4, 4.0}, {40.0, 400.0, 4000.0}}};
mdarray_double my_darr3(false, 3,2,2, **my_data3);
```

These examples show how to give default value for one, two, and three dimensions, respectively.

A pointer array corresponding to an array data can be given to the argument of the constructor. The following example shows how to give default values to the array inside an object by using the user-created function.

```
int my_function( const double *ptr[], int nx, int ny )
{
    mdarray_double my_darr2(false, nx,ny, ptr);
        :
        :
}
```

12.2 Mathematic Functions

The available mathematic functions are shown in Table 25.

The mdarray class in the prototype is the parent class of the classes shown in Table 24. Therefore, it is possible to give an object such as mdarray_double to the argument of the mdarray class, and also it is possible to receive a return value of the mdarray class by the argument of user functions such as mdarray_double.

See also the example in Tutorial §3.6.4 in which mathematic functions are used for an array.

Function Prototype	Description
`mdarray cbrt( const mdarray &obj );`	Cubic root
`mdarray sqrt( const mdarray &obj );`	Square root
`mdarray asin( const mdarray &obj );`	Arc sine
`mdarray acos( const mdarray &obj );`	Arc cosine
`mdarray atan( const mdarray &obj );`	Arc tangent
`mdarray acosh( const mdarray &obj );`	Arc hyperbolic cosine
`mdarray asinh( const mdarray &obj );`	Arc hyperbolic sine
`mdarray atanh( const mdarray &obj );`	Arc hyperbolic tangent
`mdarray exp( const mdarray &obj );`	Exponential function with base e
`mdarray exp2( const mdarray &obj );`	Exponential function with base 2
`mdarray expm1( const mdarray &obj );`	Exponent of argument minus 1
`mdarray log( const mdarray &obj );`	Natural logarithmic function
`mdarray log1p( const mdarray &obj );`	Logarithm of 1 plus argument
`mdarray log10( const mdarray &obj );`	Logarithm with base 10
`mdarray sin( const mdarray &obj );`	Sine
`mdarray cos( const mdarray &obj );`	Cosine
`mdarray tan( const mdarray &obj );`	Tangent
`mdarray sinh( const mdarray &obj );`	Hyperbolic sine
`mdarray cosh( const mdarray &obj );`	Hyperbolic cosine
`mdarray tanh( const mdarray &obj );`	Hyperbolic tangent
`mdarray erf( const mdarray &obj );`	Error function
`mdarray erfc( const mdarray &obj );`	Complementary error function
`mdarray ceil( const mdarray &obj );`	Smallest integer not less than argument
`mdarray floor( const mdarray &obj );`	Largest integer not greater than argument
`mdarray round( const mdarray &obj );`	Round a number to the nearest integer
`mdarray trunc( const mdarray &obj );`	Truncate a number to the next nearest integer towards 0
`mdarray fabs( const mdarray &obj );`	Absolute value
`mdarray hypot( const mdarray &obj, float v );`	Euclidean distance function
`mdarray hypot( const mdarray &obj, double v );`	
`mdarray hypot( float v, const mdarray &obj );`	
`mdarray hypot( double v, const mdarray &obj );`	
`mdarray hypot( const mdarray &src0, const mdarray &src1 );`	
`mdarray pow( const mdarray &obj, float v );`	Power
`mdarray pow( const mdarray &obj, double v );`	
`mdarray pow( float v, const mdarray &obj );`	
`mdarray pow( double v, const mdarray &obj );`	
`mdarray pow( const mdarray &src0, const mdarray &src1 );`	
`mdarray fmod( const mdarray &obj, float v );`	Modulo arithmetic
`mdarray fmod( const mdarray &obj, double v );`	
`mdarray fmod( float v, const mdarray &obj );`	
`mdarray fmod( double v, const mdarray &obj );`	
`mdarray fmod( const mdarray &src0, const mdarray &src1 );`	
`mdarray remainder( const mdarray &obj, float v );`	Remainder
`mdarray remainder( const mdarray &obj, double v );`	
`mdarray remainder( float v, const mdarray &obj );`	
`mdarray remainder( double v, const mdarray &obj );`	
`mdarray remainder( const mdarray &src0, const mdarray &src1 );`	

Table 25: List of Available Mathematic Functions

12.3 List of Member Functions

A list of member functions is shown in Table 26. It contains all of member functions both defined in the parent class mdarray and redefined or additionally defined in the inherited classes (such as mdarray_double).

	Funcion Name	Description	Operating Mode Support
§12.3.1	`[]`	A reference to the specified value of the element (1 dimension)	
§12.3.2	`()`	A reference to the specified value of the element (1-3 dimensions)	
§12.3.3	`=`	Substitute an array (copy the attribute, too)	
§12.3.4	`=`	Substitute a scalar value	
§12.3.5	`+=`	Add an array to itself	◯
§12.3.6	`+=`	Add a scalar value to itself	
§12.3.7	`-=`	Subtract an array from itself	◯
§12.3.8	`-=`	Subtract a scalar value from itself	
§12.3.9	`*=`	Multiply itself by an array	◯
§12.3.10	`*=`	Multiply itself by a scalar value	
§12.3.11	`/=`	Divide itself by an array	◯
§12.3.12	`/=`	Divide itself by a scalar value	
§12.3.13	`+`	Return the object that stores the result of adding an array to itself	
§12.3.14	`+`	Return the object that stores the result of adding a scalar value to itself	
§12.3.15	`-`	Return the object that stores the result of subtracting an array from itself	
§12.3.16	`-`	Return the object that stores the result of subtracting a scalar value from itself	
§12.3.17	`*`	Return the object that stores the result of multiplying itself by an array	
§12.3.18	`*`	Return the object that stores the result of multiplying itself by a scalar value	
§12.3.19	`/`	Return the object that stores the result of dividing itself by an array	
§12.3.20	`/`	Return the object that stores the result of dividing itself by a scalar value	
§12.3.21	`==`	Compare	
§12.3.22	`!=`	Compare (negative form)	
§12.3.23	`size_type()`	An integer representing a data type	
§12.3.24	`bytes()`	The number of bytes of an element	
§12.3.25	`dim_length()`	The number of dimensions of an array	
§12.3.26	`length()`	The number of elements	
§12.3.27	`byte_length()`	Total byte size of elements (in a dimension) in an array	
§12.3.28	`col_length()`	The length of the array's column	
§12.3.29	`row_length()`	The length of the array's row	
§12.3.30	`layer_length()`	The number of the layers of the array	

Table 26: List of Member Functions Available in mdarray_* Classes (cont'd)

	Funcion Name	Description	Operating Mode Support
§12.3.31	`at()`, `at_cs()`	A reference to the specified value of the element (1-3 dimensions)	
§12.3.32	`dvalue()`	The value of the element converted into the double type	
§12.3.33	`lvalue()`, `llvalue()`	The value of the element converted into the long or long long type	
§12.3.34	`default_value()` `assign_default()`	Acquire and set the initial value upon size expansion	
§12.3.35	`auto_resize()`, `set_auto_resize()`	Acquire and set the resize mode	
§12.3.36	`rounding()` `set_rounding()`	Acquire and set the rounding off possibility	
§12.3.37	`dprint()`	Output the object information to the standard error output (for user's debug)	
§12.3.38	`carray ()`, `array_ptr()`	Acquire and set the specified element's address	
§12.3.39	`get_elements ()`	Copy the array itself to the user's buffer	
§12.3.40	`put_elements ()`	Copy the array in the user's buffer to the array itself	
§12.3.41	`getdata()`	Copy the array itself to the user's buffer	
§12.3.42	`putdata()`	Copy the array in the user's buffer to the array itself	
§12.3.43	`reverse_endian()`	Reverse endian if necessary	
§12.3.44	`init()`	Initialization of the array	
§12.3.45	`assign()`	Substitute a value for an element	◯
§12.3.46	`put()`	Set a value to an arbitrary element's point	◯
§12.3.47	`swap()`	Replace values between elements	
§12.3.48	`move()`	Copy values between elements	
§12.3.49	`cpy()`	Copy values between elements (with automatic expansion)	
§12.3.50	`insert()`	Insert an element	
§12.3.51	`crop()`	Extract an element	
§12.3.52	`erase()`	Erase an element	
§12.3.53	`resize()`	Change the length of the array	
§12.3.54	`resizeby()`	Change the length of the array relatively	
§12.3.55	`increase_dim()`	Expand the number of dimensions	
§12.3.56	`decrease_dim()`	Reduce the number of dimensions	
§12.3.57	`swap()`	Replace the object by another one	
§12.3.58	`convert()`	Convert the value of the full array element	
§12.3.59	`ceil()`	Raise decimals to the next whole number in a double type value	
§12.3.60	`floor()`	Devalue decimals in a double type value	
§12.3.61	`round()`	Round off decimals in a double type value	
§12.3.62	`trunc()`	Omit decimals in a double type value	
§12.3.63	`abs()`	Absolute value of all elements	
§12.3.64	`compare()`	Compare array objects	

Table 26: List of Member Functions Available in mdarray_* Classes(cont'd)

	Funcion Name	Description	Operating Mode Support
§12.3.65	`copy()`	Copy an array into another object	
§12.3.66	`copy()`	Copy a part of an array into another object (for image data)	
§12.3.67	`cut()`	Cut all values in an array and copy them into another object	
§12.3.68	`cut()`	Cut a part of values in an array and copy them into another object (for image data)	
§12.3.69	`clean()`	Padding of existing values in an array by default ones (for image data)	
§12.3.70	`fill()`	Rewrite element values (for image data)	
§12.3.71	`add()`	Add element values (for image data)	
§12.3.72	`multiply()`	Multiply element values (for image data)	
§12.3.73	`paste()`	Paste up an array object (for image data)	
§12.3.74	`add()`	Add an array object (for image data)	
§12.3.75	`subtract()`	Subtract an array object (for image data)	
§12.3.76	`multiply()`	Multiply an array object (for image data)	
§12.3.77	`divide()`	Divide an array object (for image data)	

Table 26: List of Member Functions Available in mdarray_* Classes

12.3.1 []

NAME

`[]` — A reference to the specified value of the element (1 dimension)

SYNOPSIS

```
mdarray_type &operator[]( ssize_t idx0 );  ..................................  1
const mdarray_type &operator[]( ssize_t idx0 ) const;  ......................  2
```

DESCRIPTION

This operator returns a reference to an element specified in the square brackets. It has one argument. For multidimensional arrays, the operator `()` is used. (See §12.3.2.)

The member function 1 is available for read/write and corresponds to `at()`. The member function 2 is available for read only and corresponds to `at_cs()`.

When reading/writing a value with the member function 1 in the Automatic Resize Mode, the size of an array is resized according to the specified index. In the Manual Resize Mode, the substitution of a value into an element beyond the size of the array does not cause any error. The operation is ignored. In order to substitute a value into an element beyond the size of the array, the array size has to be extended by a member function (e.g. resize()) in advance. For more information about resize(), see §12.3.53.

When reading the element beyond the array size in the Manual Resize Mode, `NAN` is returned for floating-point values, and any one of `INDEF_UCHAR`, `INDEF_INT16`, `INDEF_INT32`, or `INDEF_INT64` is returned for integer values according to the data type of the element, respectively. The value of each `INDEF` is the minimum integer value of the data type.

Whether the member function 1 or 2 is used depends on whether the object has the “const ” attribute. The member function 1 is used for the object without the “const ” attribute, and the function 2 is automatically used with the attribute.

For information about `at()`, `at_cs()`, see §12.3.31.

PARAMETER

[I] `idx0` Subscript for the first dimension being designated as 0
([I] : input, [O] : output)

RETURN VALUE

A reference to the value of the element

EXCEPTION

The member function 1 throws an exception when it fails to allocate a local buffer in the Automatic Resize Mode.

EXAMPLE

The following code substitutes values into the mdarray_llong-class object `my_mdarr` (the data type of array is long long):

```
mdarray_llong my_mdarr;
my_mdarr[0] = 17090000;
my_mdarr[1] = 9980000;
my_mdarr[2] = 9620000;
```

12.3.2 ()

NAME

`()` — A reference to the specified value of the element (1-3 dimensions)

SYNOPSIS

```
mdarray_type &operator()( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                          ssize_t idx2 = MDARRAY_INDEF);  ..................  1
const mdarray_type &operator()( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                                ssize_t idx2 = MDARRAY_INDEF) const;  .....  2
```

DESCRIPTION

This operator returns a reference to an element specified by the index. Up to three arguments can be specified. The element index of each dimension is passed to the argument in a straightforward way for one, two, or three dimensional objects. For four or higher dimensional objects, after the array dimension of the object is reduced to three, pass the element index of the third dimension (dimension index: 2) to `idx2` to handle `n` dimensional arrays.

The member function 1 is available for read/write and corresponds to `at()`. The member function 2 is available for read only and corresponds to `at_cs()`.

When reading/writing a value with the function in the Automatic Resize Mode, the size of an array is resized according to the specified index. In the Manual Resize Mode, the substitution of a value into an element beyond the array size does not cause any error. The operation is ignored. In order to substitute a value into an element beyond the array size, the array size has to be extended by a member function (e.g. resize()) in advance. For more information about resize(), see §12.3.53.

When specifying a negative number for an element index or reading the element beyond the array size in the Manual Resize Mode, `NAN` is returned for floating-point values, and `INDEF_UCHAR`, `INDEF_INT16`, `INDEF_INT32`, or `INDEF_INT64` are returned for integer values according to the data type of the element, respectively. The value of each `INDEF` is the minimum integer value of the data type.

Whether the member function 1 or 2 is used depends on whether the object has the "const " attribute. The member function 1 is used for the object without the "const " attribute, and the function 2 is used automatically with the attribute.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

For more information about `at()`, `at_cs()`, see §12.3.31.

PARAMETER

[I] `idx0` Subscript for the first dimension being designated as as 0
[I] `idx1` Subscript for the second dimension being designated as as 1 (optional)
[I] `idx2` Subscript for the third dimension being designated as as 2 (optional)
([I] : input, [O] : output)

RETURN VALUE

A reference to the value of the element

EXCEPTION

The member function 1 throws an exception when it fails to allocate a local buffer in the Automatic Resize Mode.

EXAMPLE

The following code substitutes values into the mdarray_double-class object `my_mdarr` (the data type of array is double). A three-dimension array (3×2×1) is created:

```
mdarray_double my_mdarr;
my_mdarr(2,1,0) = 170.9;
```

12.3.3 =

NAME

= — Substitute an array (copy the attribute, too)

SYNOPSIS

```
mdarray_type &operator=(const mdarray_type &obj);  ..... 1
mdarray_type &operator=(const mdarray &obj);  ..... 2
```

DESCRIPTION

This operator copies all contents of `obj`, including the attributes such as the length of the array and resize settings to the object itself.

The argument of the member function 2 is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument. In this case, all variables are initialized to 0 and those of the derived class object are added by the operator `+=`(§12.3.5), and the attributes such as resize setting are copied.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code substitutes the mdarray_long-class object `area_mdarr` into the mdarray_llong-class object `my_mdarr` and prints the result to stdout. For more information about `length()`, see §12.3.26.

```
stdstreamio sio;

mdarray_llong my_mdarr;
mdarray_long area_mdarr;
area_mdarr[0] = 17090000;
area_mdarr[1] = 9980000;
area_mdarr[2] = 9620000;

my_mdarr = area_mdarr;
for ( size_t i=0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%lld]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [17090000]
my_mdarr value[1]...  [9980000]
my_mdarr value[2]...  [9620000]
```

12.3.4 =

NAME

= — Substitute a scalar value

SYNOPSIS

```
mdarray_type &operator=(double v);  ..... 1
mdarray_type &operator=(long long v);  ..... 2
mdarray_type &operator=(long v);  ..... 3
mdarray_type &operator=(int v);  ..... 4
```

DESCRIPTION

This operator substitutes the value (scalar value) specified by the right side of the operator (argument) into the object itself. The array size is not extended automatically, so it is necessary to set the number of elements and reserve the buffer area in advance.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code substitutes the scalar value, `125`, into the object `mdarr` with a one-dimension array and prints the result to stdout. For more information about `length()`, see §12.3.26.

```
stdstreamio sio;
mdarray_int my_mdarr(false, 2);

my_mdarr = 125;
for ( size_t i=0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%d]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [125]
my_mdarr value[1]...  [125]
```

12.3.5 +=

NAME

+= — Add an array to itself

SYNOPSIS

```
mdarray_type &operator+=(const mdarray &obj);
```

DESCRIPTION

This operator adds the object array of the mdarray (derived) class specified by the right side of the operator (argument) to the object itself. The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

In the Automatic Resize Mode, the array size is extended automatically if each dimension size of `obj` is larger than that of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

12.3.6 +=

NAME

+= — Add a scalar value to itself

SYNOPSIS

```
mdarray_type &operator+=(double v);
mdarray_type &operator+=(long long v);
mdarray_type &operator+=(long v);
mdarray_type &operator+=(int v);
```

DESCRIPTION

This operator adds the scalar value specified by the right side of the operator (argument) to all elements of the object itself. When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code adds the scalar value `50` to the mdarray_int-class object `my_mdarr` and prints the result to stdout. For more information about `length()`, see §12.3.26.

```
stdstreamio sio;

mdarray_int my_mdarr(false, 2);
my_mdarr = 25;
my_mdarr += 50;
for ( size_t i=0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%d]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [75]
my_mdarr value[1]...  [75]
```

12.3.7 -=

NAME

−= — Subtract an array from itself

SYNOPSIS

```
mdarray_type &operator-=(const mdarray &obj);
```

DESCRIPTION

This operator subtracts the object array of the mdarray (derived) class specified by the right side of the operator (argument) from the object itself. The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

In the Automatic Resize Mode, the array size is extended automatically if each dimension size of `obj` is larger than that of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code subtracts the mdarray_int-class object `subst_mdarr` from the mdarray_long-class object `my_mdarr` and prints the result to stdout. For more information about `length()`, see §12.3.26.

```
stdstreamio sio;

mdarray_long my_mdarr(false, 2);
my_mdarr = 100;

mdarray_int subst_mdarr(false, 2);
subst_mdarr[0] = 10;
subst_mdarr[1] = 20;

my_mdarr -= subst_mdarr;
for ( size_t i=0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%ld]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [90]
my_mdarr value[1]...  [80]
```

12.3.8 -=

NAME

−= — Subtract a scalar value from itself

SYNOPSIS

```
mdarray_type &operator-=(double v);
mdarray_type &operator-=(long long v);
mdarray_type &operator-=(long v);
mdarray_type &operator-=(int v);
```

DESCRIPTION

This operator subtracts the scalar value specified by the right side of the operator (argument) from all elements of the object itself. When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

12.3.9 *=

NAME

∗= — Multiply itself by an array

SYNOPSIS

```
mdarray_type &operator*=(const mdarray &obj);
```

DESCRIPTION

This operator multiplies the object itself by the object array of the mdarray (derived) class specified by the right side of the operator (argument). The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

In the Automatic Resize Mode, the array size is extended automatically if each dimension size of `obj` is larger than that of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code multiplies the mdarray_long-class object `my_mdarr` by the mdarray_int-class object `mdarrPlus` and prints the result to stdout. For more information about `length()`, see §12.3.26.

```
stdstreamio sio;

mdarray_long my_mdarr(false, 2);
my_mdarr = 50;

mdarray_int multi_mdarr;
multi_mdarr[0] = 10;
multi_mdarr[1] = 20;
multi_mdarr[2] = 30;

my_mdarr *= multi_mdarr;
for ( size_t i=0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%ld]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [500]
my_mdarr value[1]...  [1000]
```

12.3.10 *=

NAME

*= — Multiply itself by a scalar value

SYNOPSIS

```
mdarray_type &operator*=(double v);
mdarray_type &operator*=(long long v);
mdarray_type &operator*=(long v);
mdarray_type &operator*=(int v);
```

DESCRIPTION

This operator multiplies all elements of the object itself by the scalar value specified by the right side of the operator (argument). When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

PARAMETER

[I] v A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

12.3.11 /=

NAME

/= — Divide itself by an array

SYNOPSIS

```
mdarray_type &operator/=(const mdarray &obj);
```

DESCRIPTION

This operator divides the object itself by the object array of the mdarray (derived) class specified by the right side of the operator (argument). The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

In the Automatic Resize Mode, the array size is extended automatically if each dimension size of `obj` is larger than that of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code divides the mdarray_long-class object `my_mdarr` by the mdarray_int-class object `div_mdarr` and prints the result to stdout. For more information about `length()`, see §12.3.26.

```
stdstreamio sio;

mdarray_long my_mdarr(false, 2);
my_mdarr = 50;

mdarray_int div_mdarr;
div_mdarr[0] = 1;
div_mdarr[1] = 2;
div_mdarr[2] = 5;

my_mdarr /= div_mdarr;
for ( size_t i=0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%ld]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [50]
my_mdarr value[1]...  [25]
```

12.3.12 /=

NAME

/= — Divide itself by a scalar value

SYNOPSIS

```
mdarray_type &operator/=(double v);
mdarray_type &operator/=(long long v);
mdarray_type &operator/=(long v);
mdarray_type &operator/=(int v);
```

DESCRIPTION

This operator divides all elements of the object itself by the scalar value specified by the right side of the operator (argument). When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

12.3.13 +

NAME

+ — Return the object that stores the result of adding an array to itself

SYNOPSIS

```
mdarray operator+(const mdarray &obj);
```

DESCRIPTION

This operator returns the object that stores the result of adding the object array of the mdarray (derived) class specified by the right side of the operator (argument) to the object itself. The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

An object including the calculation result

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

See §3.6.4 for an example of applying operators to arrays.

12.3.14 +

NAME

+ — Return the object that stores the result of adding a scalar value to itself

SYNOPSIS

```
mdarray operator+(double v);
mdarray operator+(float v);
mdarray operator+(long long v);
mdarray operator+(long v);
mdarray operator+(int v);
```

DESCRIPTION

This operator returns the object that stores the result of adding the scalar value specified by the right side of the operator (argument) to all elements of the object itself. When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

12.3.15 -

NAME

− — Return the object that stores the result of subtracting an array from itself

SYNOPSIS

```
mdarray operator-(const mdarray &obj);
```

DESCRIPTION

This operator returns the object that stores the result of subtracting the object array of the mdarray (derived) class specified by the right side of the operator (argument) from the object itself. The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

An object including the calculation result

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

See §3.6.4 for an example of applying operators to arrays.

12.3.16 -

NAME

– — Return the object that stores the result of subtracting a scalar value from itself

SYNOPSIS

```
mdarray operator-(double v);
mdarray operator-(float v);
mdarray operator-(long long v);
mdarray operator-(long v);
mdarray operator-(int v);
```

DESCRIPTION

This operator returns the object that stores the result of subtracting the scalar value specified by the right side of the operator (argument) from all elements of the object itself. When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

12.3.17 *

NAME

∗ — Return the object that stores the result of multiplying itself by an array

SYNOPSIS

```
mdarray operator*(const mdarray &obj);
```

DESCRIPTION

This operator returns the object that stores the result of multiplying the object itself by the object array of the mdarray (derived) class specified by the right side of the operator (argument). The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from “mdarray”
([I] : input, [O] : output)

RETURN VALUE

An object including the calculation result

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

See §3.6.4 for an example of applying operators to arrays.

12.3.18 *

NAME

∗ — Return the object that stores the result of multiplying itself by a scalar value

SYNOPSIS

```
mdarray operator*(double v);
mdarray operator*(float v);
mdarray operator*(long long v);
mdarray operator*(long v);
mdarray operator*(int v);
```

DESCRIPTION

This operator returns the object that stores the result of multiplying all elements of the object itself by the scalar value specified by the right side of the operator (argument). When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `v` A real or integer scalar
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

12.3.19 /

NAME

/ — Return the object that stores the result of dividing itself by an array

SYNOPSIS

```
mdarray operator/(const mdarray &obj);
```

DESCRIPTION

This operator returns the object that stores the result of dividing the object itself by the object array of the mdarray (derived) class specified by the right side of the operator (argument). The argument is the base class (mdarray class). Thus, the object of the derived class different from itself can be specified to the argument and cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"

([I] : input, [O] : output)

RETURN VALUE

An object including the calculation result

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

See §3.6.4 for an example of applying operators to arrays.

12.3.20 /

NAME

/ — Return the object that stores the result of dividing itself by a scalar value

SYNOPSIS

```
mdarray operator/(double v);
mdarray operator/(float v);
mdarray operator/(long long v);
mdarray operator/(long v);
mdarray operator/(int v);
```

DESCRIPTION

This operator returns the object that stores the result of dividing all elements of the object itself by the scalar value specified by the right side of the operator (argument). When the data type of the argument is different from that of the object itself, cast operations are executed just like a normal scalar operation.

The returned operation mode and rounding attribute are identical to those of the object itself.

PARAMETER

[I] `v` A real or integer scalar

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

12.3.21 ==

NAME

== — Compare

SYNOPSIS

```
bool operator==(const mdarray &obj) const;
```

DESCRIPTION

This operator compares the object of mdarray (derived) class specified by the right side of the operator (argument) with the object itself. If the array size and elements of the argument `obj` are identical to those of the object itself, it returns `true`. Otherwise, it returns `false`. This member function uses compare() function(§12.3.64) internally.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

true : If the sizes and values of the elements on the arrays are identical
false : If the sizes and one of the values of the elements on the arrays are not identical

EXAMPLE

The following code compares the mdarray_uchar-class object `my_mdarr` with the mdarray_long-class object `comp_mdarr` and prints the result to stdout:

```
stdstreamio sio;

mdarray_uchar my_mdarr(false, 3);
my_mdarr = 20;

mdarray_long comp_mdarr;
comp_mdarr[0] = 20;
if (my_mdarr == comp_mdarr) {
    sio.printf("true\n");
} else {
    sio.printf("false\n");
}
```

Output:

```
false
```

12.3.22 !=

NAME

!= — Compare (negative form)

SYNOPSIS

```
bool operator!=(const mdarray &obj) const;
```

DESCRIPTION

This operator compares the object of the mdarray (derived) class specified by the right side of the operator (argument) with the object itself. If the the argument `obj` is not identical to the object itself, it returns `true`. Otherwise, it returns `false`. This member function uses the compare() function (§12.3.64) internally.

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

true : If the sizes and one of the values of the elements on the arrays are not identical
false : If the sizes and values of the elements on the arrays are identical

EXAMPLE

The following code compares the mdarray_uchar-class object `my_mdarr` with the mdarray_long-class object `comp_mdarr` and prints the result to stdout:

```
stdstreamio sio;

mdarray_uchar my_mdarr(false, 3);
my_mdarr = 20;

mdarray_long comp_mdarr;
comp_mdarr[0] = 20;
if (my_mdarr != comp_mdarr) {
    sio.printf("true\n");
} else {
    sio.printf("false\n");
}
```

Output:

```
true
```

12.3.23 size_type()

NAME

size_type() — An integer representing a data type (by data type)

SYNOPSIS

```
ssize_t size_type() const;
```

DESCRIPTION

This member function returns an integer value that corresponds to the data type of the arrays of the object itself. The values are defined in "`sli/size_types.h`" as follows:

Data Type in C	Constant Identifier	Value	Description
float	FLOAT_ZT	-4	Single-precision floating-point number
double	DOUBLE_ZT	-8	Double-precision floating-point number
fcomplex	FCOMPLEX_ZT	-7	Single-precision floating-point complex number
dcomplex	DCOMPLEX_ZT	-15	Double-precision floating-point complex number
unsigned char	UCHAR_ZT	1	Unsigned 1-byte integer
short	SHORT_ZT	System-dependent	Signed integer
int	INT_ZT	System-dependent	Signed integer
long	LONG_ZT	System-dependent	Signed integer
long long	LLONG_ZT	System-dependent	Signed integer
int16_t	INT16_ZT	2	Signed 2-byte integer
int32_t	INT32_ZT	4	Signed 4-byte integer
int64_t	INT64_ZT	8	Signed 8-byte integer
size_t	SIZE_ZT	System-dependent	Unsigned integer
ssize_t	SSIZE_ZT	System-dependent	Signed integer
bool	BOOL_ZT	System-dependent	Boolean
uintptr_t	UINTPTR_ZT	System-dependent	Unsigned integer corresponding to the address width

See Table 24 for the classes that are already available.

When the information of the data type is used in the code, use the constant identifier shown in the above table, not a raw number suc as `-4`.

RETURN VALUE

An integer to represent its type

EXAMPLE

The following code creates the mdarray_int32-class object `my_mdarr` and prints the size_type of `my_mdarr` to stdout:

```
stdstreamio sio;

mdarray_int32 my_mdarr;
sio.printf("*** my_mdarr size_type... [%zd]\n", my_mdarr.size_type());
```

Output:

```
*** my_mdarr size_type... [4]
```

12.3.24 bytes()

NAME

bytes() — The number of bytes of an element

SYNOPSIS

```
size_t bytes() const;
```

DESCRIPTION

This member function returns the byte size of an element in the array of the object itself.

RETURN VALUE
A byte length of an element

EXAMPLE
The following code creates the mdarray_double-class object `my_mdarr` and prints the byte size of an element to stdout:

```
stdstreamio sio;

mdarray_double my_mdarr;
sio.printf("*** my_mdarr bytes... [%zu]\n", my_mdarr.bytes());
```

Output:
```
*** my_mdarr bytes... [8]
```

12.3.25 dim_length()

NAME
dim_length() — The number of dimensions of an array

SYNOPSIS
```
size_t dim_length() const;
```

DESCRIPTION
This member function returns the number of dimensions for the array of the object itself.

RETURN VALUE
The number of dimensions of the array

EXAMPLE
The following code prints the number of dimensions for the array of the object `my_mdarr3dim` to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim dim... [%zu]\n", my_mdarr3dim.dim_length());
```

Output:
```
*** my_mdarr3dim dim... [3]
```

12.3.26 length()

NAME
length() — The number of elements

SYNOPSIS
```
size_t length() const;  ..... 1
size_t length( size_t dim_index ) const;  ..... 2
```

DESCRIPTION

This member function returns the total number of elements in the arrays of the object itself. When the argument is not specified, the number of elements in dimension 1 × in dim. 2 × in dim. 3 ... is returned. When `dim_index` is passed to the argument, the number of elements in the dimension with the index `dim_index` is returned. `dim_index` starts from `0`.

PARAMETER

[I] `dim_index` The integer number (≥ 0) that specifies one of the dimensions of the array ([I] : input, [O] : output)

RETURN VALUE

The number of elements

EXAMPLE

The following code prints the total number of elements in the object `my_mdarr3dim` and the number of elements in the dimension with the index 0 to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim length... [%zu]\n", my_mdarr3dim.length());
sio.printf("*** my_mdarr3dim length 1dim... [%zu]\n", my_mdarr3dim.length(0));
```

Output:

```
*** my_mdarr3dim length... [60]
*** my_mdarr3dim length 1dim... [3]
```

12.3.27 byte_length()

NAME

byte_length() — Total byte size of elements (in a dimension) in an array

SYNOPSIS

```
size_t byte_length() const;  ..... 1
size_t byte_length( size_t dim_index ) const;  ..... 2
```

DESCRIPTION

This member function returns the total byte size of arrays in the object itself. When `dim_index` is passed to the argument, the byte size of an array for the dimension with the index `dim_index` is returned.

PARAMETER

[I] `dim_index` The integer number (≥ 0) that specifies one of the dimensions of the array ([I] : input, [O] : output)

RETURN VALUE

The whole byte size of the array or the byte size of the elements in the specified dimension

EXAMPLE

The following code prints the total byte size of the arrays in a three-dimension array `my_mdarr3dim` and the byte size of an array for the third dimension (dimension index: 2) to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim byte_length... [%zu]\n",
            my_mdarr3dim.byte_length());
sio.printf("*** my_mdarr3dim byte_length 3dim... [%zu]\n",
            my_mdarr3dim.byte_length(2));
```

Output:

```
*** mdarr3dim byte_length... [240]
*** mdarr3dim byte_length 3dim... [20]
```

12.3.28 col_length()

NAME

length() — The length of the array's column

SYNOPSIS

```
size_t col_length() const;
```

DESCRIPTION

This member function returns the length of the columns for the array of the object itself.

RETURN VALUE

A column length of the array

EXAMPLE

The following code prints the length of the columns for a three-dimension array `my_mdarr3dim` to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim col... [%zu]\n", my_mdarr3dim.col_length());
```

Output:

```
*** my_mdarr3dim col... [3]
```

12.3.29 row_length()

NAME

row_length() — The length of the array's row

SYNOPSIS

```
size_t row_length() const;
```

DESCRIPTION

This member function returns the length of the rows for the array of the object itself.

RETURN VALUE

A row length of the array

EXAMPLE

The following code prints the length of the rows for a three-dimension array `my_mdarr3dim` to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim row... [%zu]\n", my_mdarr3dim.row_length());
```

Output:

```
*** my_mdarr3dim row... [4]
```

12.3.30 layer_length()

NAME

layer_length() — The number of the layers of the array

SYNOPSIS

```
size_t layer_length() const;
```

DESCRIPTION

This member function returns the length of the layers for the array of the object itself. When the dimension of the array is 1 or 2, 1 is returned. When the dimension of the array is 3 or more, after the array dimension is reduced to three, the length of the layers for the third dimension (dimension index: 2) is returned.

RETURN VALUE

The number of dimensions of the array

EXAMPLE

The following code prints the length of the layers for a three-dimension array `my_mdarr3dim` to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim layer... [%zu]\n", my_mdarr3dim.layer_length());
```

Output:

```
*** my_mdarr3dim layer... [5]
```

12.3.31 at(), at_cs()

NAME

at(), at_cs() — A reference to the specified value of the element (1-3 dimensions)

SYNOPSIS

```
type &at( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
          ssize_t idx2 = MDARRAY_INDEF );  ..................................  1
const type &at( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                ssize_t idx2 = MDARRAY_INDEF ) const;  ......................  2
const type &at_cs( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                   ssize_t idx2 = MDARRAY_INDEF ) const;  ..................  3
```

DESCRIPTION

This member function sets/gets the element specified by the arguments `idx0`, `idx1`, and `idx2` to/from the array.

The member function 1 is available for read/write. The member functions 2 and 3 are available for read only.

When reading/writing a value with the member function 1 in the Automatic Resize Mode, the size of an array is resized according to the specified index. In the Manual Resize Mode, the substitution of a value into an element beyond the array size does not cause any error. The operation is ignored. In order to substitute a value into an element beyond the array size, the array size has to be extended by a member function (e.g. resize()) in advance. For more information about resize(), see §12.3.53.

When reading the element beyond the array size in Manual Resize Mode, `NAN` is returned for floating-point numbers, and any one of `INDEF_UCHAR`, `INDEF_INT16`, `INDEF_INT32`, or `INDEF_INT64` is returned for integer numbers according to the data type of the element, respectively. The value of each `INDEF` is the minimum integer value of the data type.

For the function at(), whether the member function 1 or 2 is used depends on whether the object has the "const " attribute. The member function 1 is used for the object without the "const " attribute, and the function 2 is used with the attribute automatically.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I] `idx0` Subscript for the first dimension being designated as 0
[I] `idx1` Subscript for the second dimension being designated as 1 (optional)
[I] `idx2` Subscript for the third dimension being designated as 2 (optional)
([I] : input, [O] : output)

RETURN VALUE

A reference to the values of the elements

EXCEPTION

The member function 1 throws an exception when it fails to allocate a local buffer in the Automatic Resize Mode.
All member functions throw an exception when the size of each element in this array is greater than that of their return value.

EXAMPLE

The following code sets values to the elements of the mdarray_float-class object `my_fmdarr` via a member function at() and prints the values of elements to stdout by the at():

```
stdstreamio sio;

mdarray_float my_fmdarr;

my_fmdarr.at(0) = 1000.1;
my_fmdarr.at(1) = 2000.2;
for ( size_t i = 0 ; i < my_fmdarr.length() ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%5.1f]\n", i, my_fmdarr.at(i));
}
```

Output:

```
my_fmdarr value[0]...  [1000.1]
my_fmdarr value[1]...  [2000.2]
```

12.3.32 dvalue()

NAME

dvalue() — The value of the element converted into the double type

SYNOPSIS

```
double dvalue( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
               ssize_t idx2 = MDARRAY_INDEF ) const;
```

DESCRIPTION

This member function casts the element in the array of the object itself to a double-precision floating-point value and returns it. When the specified index exceeds the array size, `NAN` is returned.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I] `idx0` Subscript for the first dimension being designated as 0
[I] `idx1` Subscript for the second dimension being designated as 1 (optional)
[I] `idx2` Subscript for the third dimension being designated as 2 (optional)
([I] : input, [O] : output)

RETURN VALUE

A value converted to the double type	:	Normal end
`NAN`(error)	:	When the type of the element is not supported When the index of the elements exceeding the size of the array is specified

EXAMPLE

The following code sets a value to the mdarray_float-class object `my_mdarry` and gets the value as a double-precision floating-point number, and prints it to stdout:

```
stdstreamio sio;

mdarray_float my_mdarry;
my_mdarry[0] = 123.456;
sio.printf("my_mdarry dvalue... [%6.3f]\n", my_mdarry.dvalue(0));
```

Output:

```
my_mdarry dvalue...  [123.456]
```

12.3.33 lvalue(), llvalue()

NAME

lvalue(), llvalue() — The value of the element converted into the long or long long type

SYNOPSIS

```
long lvalue( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
             ssize_t idx2 = MDARRAY_INDEF ) const;  ..........................  1
long long llvalue( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                   ssize_t idx2 = MDARRAY_INDEF ) const;  ...................  2
```

DESCRIPTION

This member function casts the element in the array of the object itself to a long or long long value and returns it.

When the data type of the array is floating-point, the value is truncated to the whole number by default. In order to round it to the whole number, it requires the use of the set_rounding() function in advance. For more information about set_rounding(), see §12.3.36.

When the specified index exceeds the array size, `INDEF_LONG` or `INDEF_LLONG` is returned, respectively. The value of each `INDEF` is the minimum integer value of the data type.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I] `idx0` Subscript for the first dimension being designated as 0
[I] `idx1` Subscript for the second dimension being designated as 1 (optional)
[I] `idx2` Subscript for the third dimension being designated as 2 (optional)
([I] : input, [O] : output)

RETURN VALUE

A value converted to the long or long long type	:	Normal end
`INDEF_LONG` or `INDEF_LLONG`	:	When the type of the element is not supported When the index of the elemnts exceeding the size of the array is specified

EXAMPLE

The following code sets a value to the mdarray_float-class object `my_mdarry` and gets the value as a long number and as a long long number, and prints them to stdout:

```
stdstreamio sio;

mdarray_float my_mdarry;
my_mdarry[0] = 123.556;
sio.printf("my_mdarry lvalue... [%ld]\n", my_mdarry.lvalue(0));
my_mdarry.set_rounding(true);
sio.printf("my_mdarry llvalue... [%lld]\n", my_mdarry.llvalue(0));
```

Output:

```
my_mdarry lvalue...  [123]
my_mdarry llvalue...  [124]
```

12.3.34 default_value(), assign_default()

NAME

default_value(), assign_default() — Acquire and set the initial value upon size expansion

SYNOPSIS

```
type default_value() const;  ..... 1
mdarray_type &assign_default( type value );  ..... 2
```

DESCRIPTION

The member function 1 returns the initial value for size extension.

The member function 2 sets up the initial value for size extension. The value does not apply to the existing elements. It becomes valid when the array size is extended.

RETURN VALUE

The member function 1 returns the initial value when the size of the array is expanded.

The member function 2 returns a reference to itself.

EXCEPTION

The function 2 throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code prints the initial value of the mdarray_float-class object `my_mdarr` for size extension to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr;
my_mdarr.assign_default(50);
sio.printf("*** my_mdarr defval... [%f]\n",
           my_mdarr.default_value());
```

Output:

```
*** my_mdarr defval... [50.000000]
```

12.3.35 auto_resize(), set_auto_resize()

NAME

auto_resize(), set_auto_resize() — Acquire and set the resize mode

SYNOPSIS

```
bool auto_resize() const;  ..................................................... 1
mdarray_type &set_auto_resize( bool tf );  ..................................... 2
```

DESCRIPTION

The member function 1 returns the resize mode by boolean type. The member function 2 sets up the resize mode by boolean type.

The Automatic Resize Mode corresponds to true (=1) and the Manual Resize Mode corresponds to false (=0).

See the Operation Mode Support box in Table 26 to find out which member functions support the operation modes.

RETURN VALUE

The member function 1 returns an operation mode (or true in the Automatic Resize Mode).

The member function 2 returns a reference to itself.

EXAMPLE

The following code creates `mdarr0dim` in the Automatic Resize Mode and creates `mdarr3dim` in the Manual Resize Mode, and prints the resize modes of the arrays to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr0dim;
sio.printf("*** my_mdarr0dim auto_resize... [%d]\n",
            (int)(my_mdarr0dim.auto_resize()));
mdarray_float my_mdarr3dim(false, 3, 4, 5);
sio.printf("*** my_mdarr3dim auto_resize... [%d]\n",
            (int)(my_mdarr3dim.auto_resize()));
```

Output:

```
*** my_mdarr0dim auto\_resize... [1]
*** my_mdarr3dim auto\_resize... [0]
```

12.3.36 rounding(), set_rounding()

NAME

rounding(), set_rounding() — Acquire and set the rounding off possibility

SYNOPSIS

```
bool rounding() const;  ..................................................... 1
mdarray_type &set_rounding( bool tf );  ..................................... 2
```

DESCRIPTION

The member function 2 sets up the rounding mode. Either the floating-point numbers are truncated or rounded to the whole number in some high-level member functions. The member function 1 returns true in the round mode and false in the truncate mode.

Upon creation of an object, it is set to the truncate mode.

When objects are copied by the operator "=" or the init() member function, the rounding attribution is also copied. For more information about init(), see §12.3.44.

The member functions that support the rounding attribution are: lvalue(), llvalue() (§12.3.33), assign_default() (§12.3.34), assign() (§12.3.45), and all member functions for images.

RETURN VALUE

The member function 1 returns an attribute of the operation on rounding.

The member function 2 returns a reference to itself.

EXAMPLE

The following code creates the mdarray_llong-class object `my_mdarr` and sets a real number twice before and after setting the rounding mode. Then the code prints the substituted values to stdout for confirmation:

```
stdstreamio sio;

mdarray_llong my_mdarr;
my_mdarr.assign(1.618, 0);
sio.printf("my_mdarr value[0]... [%lld]\n", my_mdarr[0]);

my_mdarr.set_rounding(true);
my_mdarr.assign(1.618, 1);
sio.printf("my_mdarr value[1]... [%lld]\n", my_mdarr[1]);
```

Output:

```
my_imdarr value[0]...  [1]
my_imdarr value[1]...  [2]
```

12.3.37 dprint()

NAME

dprint() — Output of the object information to the standarderr output (for user's debug)

SYNOPSIS

```
void dprint() const;
```

DESCRIPTION

This member function prints the information of the object itself to stderr.

This is a function for debugging a user program.

EXAMPLE

The following code prints the information on the object `my_array` to stderr. The address of the object in `[]` is system-dependent.

```
mdarray_int my_array (false, 3,2,1);
my_array (2,0,0) = 100;
my_array (0,1,0) = 200;
my_array.dprint();
```

Output:

```
sli::mdarray[obj=0x7fbffff630, sz_type=4, dim=(3,2,1)] = {
 { { 0,0,100 },
   { 200,0,0 } }
}
```

12.3.38 carray (), array_ptr()

NAME

carray (), array_ptr() — Acquire the specified element's address

SYNOPSIS

```
const type *carray () const;  ..... 1
const type *carray ( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                     ssize_t idx2 = MDARRAY_INDEF ) const;  ..... 2
type *array_ptr();  ..... 3
type *array_ptr( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                 ssize_t idx2 = MDARRAY_INDEF );  ..... 4
const type *array_ptr() const;  ..... 5
const type *array_ptr( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                       ssize_t idx2 = MDARRAY_INDEF ) const;  ..... 6
const type *array_ptr_cs() const;  ..... 7
const type *array_ptr_cs( ssize_t idx0, ssize_t idx1 = MDARRAY_INDEF,
                          ssize_t idx2 = MDARRAY_INDEF ) const;  ..... 8
```

DESCRIPTION

These member functions get the address of the element specified by the index in the array of the object itself. Member functions 1, 2, and 5 through 8 get the address for read only.

For the function array_ptr(), whether the member functions 3, 4 or 5, 6 are used depends on whether the oobject has the "const " attribute. The member function 3 or 4 is used for the object without the "const " attribute, and the function 5 or 6 is used with the attribute automatically.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I] `idx0` Subscript for the first dimension being designated as 0
[I] `idx1` Subscript for the second dimension being designated as 1 (optional)
[I] `idx2` Subscript for the third dimension being designated as 2 (optional)
([I] : input, [O] : output)

RETURN VALUE

An address of the specified element

EXAMPLE

The following code gets the address of the value at (0, 1) in the object `my_fmdarr` with a two-dimension array and prints the value to stdout:

```
stdstreamio sio;
mdarray_float my_fmdarr(false, 2,2);
my_fmdarr(0,0) = 1000;
my_fmdarr(1,0) = 2000;
my_fmdarr(0,1) = 3000;
my_fmdarr(1,1) = 4000;

const float *mycarray_ptr = my_fmdarr.carray (0, 1);
sio.printf("*** my_fmdarr carray[0] ---> [%f] *** \n",  mycarray_ptr[0]);
```

Output:

```
*** my_fmdarr carray[0] ---> [3000.000000] ***
```

12.3.39 get_elements ()

NAME

get_elements () — Copy the array itself to the user's buffer

SYNOPSIS

```
ssize_t get_elements ( type *dest_buf, size_t elem_size,
                       ssize_t idx0 = 0, ssize_t idx1 = MDARRAY_INDEF,
                       ssize_t idx2 = MDARRAY_INDEF ) const;
```

DESCRIPTION

This member function copies the contents of the array for the object itself to the user buffer specified by `dest_buf`. The size of the buffer `elem_size` is set by the number of elements. The source is specified by the arguments `idx0`, `idx1`, and `idx2`.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[O]	`dest_buf`	Address of user's buffer
[I]	`elem_size`	The number of elements to be copied
[I]	`idx0`	Subscript of this array for its first dimension being designated as 0 (optional)
[I]	`idx1`	Subscript of this array for its second dimension being designated as 1 (optional)
[I]	`idx2`	Subscript of this array for its third dimension being designated as 2 (optional)

([I] : input, [O] : output)

RETURN VALUE

The number of the elements copied when the user buffer length is enough

EXCEPTION

The function throws an exception when it detects memory corruption.

EXAMPLE

The following code copies the contents of the object `my_fmdarr` with a two-dimension array to the user buffer `myfloat` and prints the values of elements in `myfloat` to stdout for confirmation:

```
stdstreamio sio;

float my_data[] = {1000, 2000, 3000, 4000};
mdarray_float my_fmdarr(false, 2,2, my_data);

float myfloat[4];
my_fmdarr.get_elements (myfloat, sizeof(myfloat)/sizeof(float));
for ( int i = 0 ; i < sizeof(myfloat)/sizeof(float) ; i++ ) {
    sio.printf("myfloat value[%d]... [%f]\n", i, myfloat[i]);
}
```

Output:

```
myfloat_ptr value[0]...  [1000.000000]
myfloat_ptr value[1]...  [2000.000000]
myfloat_ptr value[2]...  [3000.000000]
myfloat_ptr value[3]...  [4000.000000]
```

12.3.40 put_elements ()

NAME

put_elements () — Copy the array in the user's buffer to the array itself

SYNOPSIS

```
ssize_t put_elements ( const type *src_buf, size_t elem_size, ssize_t idx0 = 0,
                       ssize_t idx1 = MDARRAY_INDEF, ssize_t idx2 = MDARRAY_INDEF );
```

DESCRIPTION

This member function copies the contents of the user buffer specified by `src_buf` to the array for the object itself. The size of the buffer `elem_size` is set by the number of elements. The destination is specified by the arguments `idx0`, `idx1`, and `idx2`.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I]	`src_buf`	Address of the user's buffer
[I]	`elem_size`	Number of elements to be copied
[I]	`idx0`	Subscript of this array for its first dimension being designated as 0 (optional)
[I]	`idx1`	Subscript of this array for its second dimension being designated as 1 (optional)
[I]	`idx2`	Subscript of this array for its third dimension being designated as 2 (optional)

([I] : input, [O] : output)

RETURN VALUE

The number of the elements copied when the user buffer length is enough

EXCEPTION

The function throws an exception when it detects memory corruption.

EXAMPLE

The following code copies the contents of the user buffer `my_float` to the object `my_fmdarr` with a two-dimension array and prints the values of elements in `my_fmdarr` to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr(false, 2,2);

float my_float[] = {1000, 2000, 3000, 4000};
my_fmdarr.put_elements (my_float, sizeof(my_float)/sizeof(float));

for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
  for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
    sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
                i, j, my_fmdarr(i, j));
  }
}
```

Output:

```
my_fmdarr value(0,0)...  [1000.000000]
my_fmdarr value(1,0)...  [2000.000000]
my_fmdarr value(0,1)...  [3000.000000]
my_fmdarr value(1,1)...  [4000.000000]
```

12.3.41 getdata()

NAME

getdata() — Copy the array itself to the user's buffer

SYNOPSIS

```
ssize_t getdata( void *dest_buf, size_t buf_size, ssize_t idx0 = 0,
                 ssize_t idx1 = MDARRAY_INDEF, ssize_t idx2 = MDARRAY_INDEF ) const;
```

DESCRIPTION

This member function copies the contents of the array for the object itself to the user buffer specified by `dest_buf`. The size of the buffer `buf_size` is set by the byte unit. The source is specified by the arguments `idx0`, `idx1`, and `idx2`.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[O]	`dest_buf`	Address of user's buffer
[I]	`buf_size`	Size of the buffer in bytes
[I]	`idx0`	Subscript of this array for its first dimension being designated as 0 (optional)
[I]	`idx1`	Subscript of this array for its second dimension being designated as 1 (optional)
[I]	`idx2`	Subscript of this array for its third dimension being designated as 2 (optional)

([I] : input, [O] : output)

RETURN VALUE

A size of the buffer copied when the user buffer size (buf_size) is enough

EXCEPTION

The function throws an exception when memory corruption is detected.

EXAMPLE

The following code copies the contents of the object `my_fmdarr` with a two-dimension array to the user buffer `myfloat` and prints the values of elements in `myfloat` to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr(false, 2,2);
my_fmdarr(0,0) = 1000;
my_fmdarr(1,0) = 2000;
my_fmdarr(0,1) = 3000;
my_fmdarr(1,1) = 4000;

float myfloat[4];
my_fmdarr.getdata((void *)myfloat, sizeof(myfloat));
for ( int i = 0 ; i < sizeof(myfloat)/sizeof(float) ; i++ ) {
    sio.printf("myfloat value[%d]... [%f]\n", i, myfloat[i]);
}
```

Output:

```
myfloat value[0]...  [1000.000000]
myfloat value[1]...  [2000.000000]
myfloat value[2]...  [3000.000000]
myfloat value[3]...  [4000.000000]
```

12.3.42 putdata()

NAME

putdata() — Copy the array in the user's buffer to the array itself

SYNOPSIS

```
ssize_t putdata( const void *src_buf, size_t buf_size, ssize_t idx0 = 0,
                 ssize_t idx1 = MDARRAY_INDEF, ssize_t idx2 = MDARRAY_INDEF );
```

DESCRIPTION

This member function copies the contents of the user buffer specified by `src_buf` to the array for the object itself. The size of the buffer `buf_size` is set by the byte unit. The destination is specified by the arguments `idx0`, `idx1`, and `idx2`.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I]	`src_buf`	Address of user's buffer
[I]	`buf_size`	Size of the user's buffer in bytes
[I]	`idx0`	Subscript of this array for its first dimension being designated as 0 (optional)
[I]	`idx1`	Subscript of this array for its second dimension being designated as 1 (optional)
[I]	`idx2`	Subscript of this array for its third dimension being designated as 2 (optional)

([I] : input, [O] : output)

RETURN VALUE

A size of the buffer copied when the user buffer size (buf_size) is enough

EXCEPTION

The function throws an exception when it detects memory corruption.

EXAMPLE

The following code copies the contents of the user buffer `my_float` to the object `my_fmdarr` with a two-dimension array and prints the values of elements in `my_fmdarr` to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr(false, 2,2);

float my_float[] = {1000, 2000, 3000, 4000};
my_fmdarr.putdata((const void *)my_float, sizeof(my_float));

for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
  for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
    sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
                i, j, my_fmdarr(i, j));
  }
}
```

Output:

```
my_fmdarr value(0,0)...  [1000.000000]
my_fmdarr value(1,0)...  [2000.000000]
my_fmdarr value(0,1)...  [3000.000000]
my_fmdarr value(1,1)...  [4000.000000]
```

12.3.43 reverse_endian()

NAME

reverse_endian() — Reverse endian if necessary

SYNOPSIS

```
mdarray_type &reverse_endian( bool is_little_endian );  ..................  1
mdarray_type &reverse_endian( bool is_little_endian,
                              size_t begin, size_t length );  ............  2
```

DESCRIPTION

This member function is called to save the array of the object itself as a binary data in a file or load a binary data in a file to the array of the object itself.

To save data in a file, convert the endian of the data to the appropriate form for storing by this member function. Next, write the content by the stream writing function with the address obtained from the function such as carray () (§12.3.38). Then call this function again for turning back the endian.

To load data from a file, read the content by the stream reading function with the address obtained from the functions such as array_ptr() (§12.3.38). Then convert the endian to the appropriate form for the system by this member function.

In both cases shown above, if the data to be stored on a file is big endian, the first argument is set to `false` (if little endian, set to `true`).

For this member function, users do not have to be conscious of the difference in system architecture. For instance, a user specifies `false` to the argument `is_little_endian` and calls this function so that a data in big endian is saved in a file. If the machine is a big endian system, the inversion process is not executed in practice. (With a little endian system, the inversion process is executed.) Next, the user saves binary data in the object in a file in a straightforward way, and the binary file in the specified byte order is created. And then, the user calls this member function with the same arguments again in order to turn back the endian if it was inverted.

This means that to save in a file this member function must be called twice with the same arguments.

Setting `begin` and `length` arguments allows a partial endian conversion of array elements.

PARAMETER

[I]	`is_little_endian`	True when the ordering of data in a computer's memory should be little endian after one conversion
[I]	`begin`	Starting position of elements to be converted (0-indexed)
[I]	`length`	Length of elements to be converted

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code outputs the binary file that contains the data of the object `my_mdarr` with a two-dimension array in big endian:

```
stdstreamio sio;

mdarray_int my_mdarr(false, 2,2);
```

```
my_mdarr(0,0) = 10;
my_mdarr(1,0) = 20;
my_mdarr(0,1) = 30;
my_mdarr(1,1) = 40;

my_mdarr.reverse_endian(false);
const void *mydata_ptr = my_mdarr.data_ptr();

if ( fio.openf("w", "%s", "binary.dat") < 0 ) {
  // Error Handling
}
if ( fio.write(mydata_ptr, my_mdarr.byte_length()) < 0 ) {
  // Error Handling
}
my_mdarr.reverse_endian(false);

fio.close();
```

Output:
the contents of `binary.dat`:

```
"00 00 00 0A"
"00 00 00 14"
"00 00 00 1E"
"00 00 00 28"
```

For more information about endian conversion, see §3.6.10.

12.3.44 init()

NAME

init() — Initialize the array

SYNOPSIS

```
mdarray_type &init();  ..... 1
mdarray_type &init( bool auto_resize );  ..... 2
mdarray_type &init( bool auto_resize,
                    const size_t naxisx[], size_t ndim );  ..... 3
mdarray_type &init( bool auto_resize, size_t naxis0 );  ..... 4
mdarray_type &init( bool auto_resize, size_t naxis0, size_t naxis1 );  .... 5
mdarray_type &init( bool auto_resize, size_t naxis0, size_t naxis1,
                    size_t naxis2 );  ..... 6
mdarray_type &init( bool auto_resize, size_t naxis0,
                    const type vals[] );  ..... 7
mdarray_type &init( bool auto_resize, size_t naxis0, size_t naxis1,
                    const type vals[] );  ..... 8
mdarray_type &init( bool auto_resize, size_t naxis0, size_t naxis1,
                    const type *const vals[] );  ..... 9
mdarray_type &init( bool auto_resize, size_t naxis0, size_t naxis1,
                    size_t naxis2, const type vals[] );  ..... 10
mdarray_type &init( bool auto_resize, size_t naxis0, size_t naxis1,
```

DESCRIPTION

This member function initializes the array of the object itself.

The member function 1 initializes the object with the array size 0. The operation mode is set to the Automatic Resize Mode.

For the member functions 2 to 11, the operation mode is specified to the first argument `auto_resize` and the size of the arrays and the address of the arrays for initialization are specified to the rest of the arguments.

The member function 12 copies all the contents and attributes of `obj` to the object itself.

For the member functions 1 to 11, arguments are passed to them just like when creating objects (constructor). See §12.1 for the arguments and operation modes in creating objects.

PARAMETER

[I]	`auto_resize`	True if you want to use the function in the Automatic Resize Mode
[I]	`ndim`	Number of dimensions of the array
[I]	`naxisx[]`	Number of elements along each dimension
[I]	`naxis0`	Number of elements along the first dimension (dimension 0)
[I]	`naxis1`	Number of elements along the second dimension (dimension 1)
[I]	`naxis2`	Number of elements along the third dimension (dimension 2)
[I]	`vals`	the address of an input array or a pointer array
[I]	`obj`	A reference to an input object

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code initializes the object `my_mdarr` with the 2×3 array and prints the length of the dimensions to stdout:

```
stdstreamio sio;

mdarray_int my_mdarr;
my_mdarr.init(false, 2,3);
sio.printf("*** my_mdarr 0 dim length ====> [%zu] *** \n",
           my_mdarr.length(0));
sio.printf("*** my_mdarr 1st dim length ====> [%zu] *** \n",
           my_mdarr.length(1));
```

Output:

```
*** my_mdarr 0 dim length ====> [2] ***
*** my_mdarr 1st dim length ====> [3] ***
```

12.3.45 assign()

NAME

assign() — Substitute a value for an element

SYNOPSIS

```
mdarray_type &assign( double value, ssize_t idx0,
                      ssize_t idx1 = MDARRAY_INDEF, ssize_t idx2 = MDARRAY_INDEF );
```

DESCRIPTION

This member function assigns a value to the element specified by `idx`*n* in the array of the object itself. When a floating-point number is assigned to an element with the data type integer, it is truncated by default. To round it, call set_rounding() in advance. For more information about set_rounding(), see §12.3.36.

In the Automatic Resize Mode, the size of the arrays are resized according to the specified element index automatically.

In the Manual Resize Mode, assigning a value to an element beyond the array size does not cause any error. The operation is ignored. In order to assign a value to an element beyond the array size, the array size has to be extended by the member function resize() in advance. For more information about resize(), see §12.3.53.

Do not specify `MDARRAY_INDEF` for an argument explicitly.

PARAMETER

[I] `value` A real scalar in double precision
[I] `idx0` Subscript for the first dimension being designated as 0
[I] `idx1` Subscript for the second dimension being designated as 1 (optional)
[I] `idx2` Subscript for the third dimension being designated as 2 (optional)

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer in the Automatic Resize Mode.

EXAMPLE

The following code sets the value to the element specified by index 1 in the mdarray_float-class object `my_mdarr` and prints the values of elements to stdout:

```
stdstreamio sio;

mdarray_float my_mdarr;
my_mdarr.assign(200.0, 1);
for ( size_t i = 0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr lvalue[%zu]... [%ld]\n", i, my_mdarr.lvalue(i));
}
```

Output:

```
my_mdarr lvalue[0]...  [0]
my_mdarr lvalue[1]...  [200]
```

12.3.46 put()

NAME

put() — Set a value to an arbitrary element's point

SYNOPSIS

```
mdarray_type &put( type value, ssize_t idx, size_t len );  ..... 1
mdarray_type &put( type value,
                   size_t dim_index, ssize_t idx, size_t len );  ..... 2
```

DESCRIPTION

This member function puts `len` straight `value`(s) to the element index `idx` of the array in the object itself. The element index and the dimension index start from 0.

Any values can be set to `idx` and `len`. In the Automatic Resize Mode, if the length of the array in the object is lower than the specified argument, the array is resized automatically. The additional part in which the value is not set is filled with the default value. In the Automatic Resize Mode, the writing operation is not executed for the elements beyond the array size.

The member function 1 writes `len` `value`(s) to elements from `idx` in the array sequentially.

The member function 2 writes `len` `value`(s) to elements from `idx` in the dimension index `dim_index` of the arrays in the object sequentially. When `dim_index` is 1 or over, values are written to all elements in the lower dimensions.

PARAMETER

[I]	`value`	A given scalar to be written to subarray of this object
[I]	`idx`	Subscript that specifies the first element of the subarray
[I]	`len`	Number of elements in the subarray along one dimension
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer in the Automatic Resize Mode.

EXAMPLE

The following code puts two values to the array elements starting with index 1 in the object `my_smdarr` with a one-dimension array and prints the values of elements to stdout:

```
stdstreamio sio;

mdarray_short my_smdarr(false, 3);
my_smdarr.put(12, 1,2);
for ( size_t i = 0 ; i < my_smdarr.length() ; i++ ) {
    sio.printf("my_smdarr value[%zu]... [%hd]\n", i, my_smdarr[i]);
}
```

Output:

```
my_smdarr value[0]...  [0]
my_smdarr value[1]...  [12]
my_smdarr value[2]...  [12]
```

12.3.47 swap()

NAME

swap() — Replace values between elements

SYNOPSIS

```
mdarray_type &swap( ssize_t idx_src, size_t len, ssize_t idx_dst );  ..... 1
mdarray_type &swap( size_t dim_index,
                    ssize_t idx_src, size_t len, ssize_t idx_dst );  ..... 2
```

DESCRIPTION

This member function swaps values in the array of the object itself.

The member function 1 swaps `len` elements from the index `idx_src` for `len` elements from the index `idx_dst`. If `idx_dst + len` exceeds the size of the array, the process is executed for up to the size of the array.

The member function 2 swaps `len` elements from the index `idx_src` in the dimension with the index `dim_index` for `len` elements from the index `idx_dst`. If `idx_dst + len` exceeds the size of the array, the process is executed for up to the size of the array.

When the area for swapping is overlapped, only the non-overlapped area in the source area is swapped.

PARAMETER

[I]	`idx_src`	Subscript specifying the first element of one of the two subarrays to be swapped with each other
[I]	`len`	Number of elements in the subarray
[I]	`idx_dst`	Subscript specifying the first element of the other subarray
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code swaps one element specified by index 0 in the second dimension (dimension index: 1) for the element with the index 1 in the mdarray_uchar-class object `my_cmdarr` and prints the values of elements to stdout:

```
stdstreamio sio;

unsigned char my_char[] = {51, 52, 101, 102};
mdarray_uchar my_cmdarr(false, 2,2, my_char);

my_cmdarr.swap( 1, 0, 1, 1 );
for ( size_t j = 0 ; j < my_cmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_cmdarr.length(0) ; i++ ) {
        sio.printf("my_cmdarr value(%zu,%zu)... [%hhu]\n",
                   i, j, my_cmdarr(i, j));
    }
}
```

Output:

```
my_cmdarr value(0,0)...  [101]
```

```
my_cmdarr value(1,0)...  [102]
my_cmdarr value(0,1)...  [51]
my_cmdarr value(1,1)...  [52]
```

12.3.48 move()

NAME

move() — Copy values between elements

SYNOPSIS

```
mdarray_type &move( ssize_t idx_src, size_t len, ssize_t idx_dst,
                    bool clr );  ..................................... 1
mdarray_type &move( size_t dim_index, ssize_t idx_src, size_t len, ssize_t idx_dst,
                    bool clr );  ..................................... 2
```

DESCRIPTION

This member function moves values in the array of the object itself.

When `false` is passed to the argument `clr`, the values of the source remain. For `true`, the values of the source do not remain and they are filled with the default values. If the value exceeding the existing array size is specified to the argument `idx_dst`, the size is not changed. This operation is different from that of the member function cpy(). (See §12.3.49.)

For the member function 1, the moving operation is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the moving operation is applied to the elements in the dimension with the index `dim_index`.

PARAMETER

[I]	`idx_src`	Subscript specifying the first element of an input subarray in this object
[I]	`len`	Number of elements in the input subarray
[I]	`idx_dst`	Subscript specifying the first element of another subarray to which the input subarray is written
[I]	`clr`	True if the contents of the input subarray may be lost
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code copies values inside the array of the mdarray_uchar-class object `my_cmdarr` and prints the values of elements to stdout for confirmation. The values of the source are cleared.

```
stdstreamio sio;

mdarray_uchar my_cmdarr(false, 3);
my_cmdarr[0] = 99;
my_cmdarr[1] = 98;
my_cmdarr[2] = 97;
my_cmdarr.move( 2, 1, 0, true );
for ( size_t i = 0 ; i < my_cmdarr.length() ; i++ ) {
    sio.printf("my_cmdarr value[%zu]... [%hhu]\n",
```

```
                    i, my_cmdarr[i]);
    }
```

Output:

```
my_cmdarr value[0]...  [97]
my_cmdarr value[1]...  [98]
my_cmdarr value[2]...  [0]
```

12.3.49 cpy()

NAME

cpy() — Copy values between elements (with automatic expansion)

SYNOPSIS

```
mdarray_type &cpy( ssize_t idx_src, size_t len, ssize_t idx_dst,
             bool clr );  ..................................................... 1
mdarray_type &cpy( size_t dim_index, ssize_t idx_src, size_t len, ssize_t idx_dst,
             bool clr );  ..................................................... 2
```

DESCRIPTION

This member function copies values inside the array of the object itself.

When `false` is passed to the argument `clr`, the values of the source remain. For `true`, the values of the source do not remain and they are filled with the default values. If `idx_dst` + `len` exceeds the existing array size, the size is extended automatically.

For the member function 1, the copying operation is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the copying operation is applied to the elements in the dimension with the index `dim_index`.

PARAMETER

[I]	`idx_src`	Subscript specifying the first element of an input subarray
[I]	`len`	Number of elements in the input subarray
[I]	`idx_dst`	Subscript specifying the first element of another subarray to which the input subarray is written
[I]	`clr`	True if the contents of the input subarray may be lost
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code copies values inside the array of the mdarray_llong-class object `my_lmdarr` and prints the values of elements to stdout for confirmation. The values of the source remain:

```
stdstreamio sio;

mdarray_llong my_lmdarr;
my_lmdarr[0] = -2147483646;
my_lmdarr[1] = 2147483647;
```

```
my_lmdarr.cpy(1, 1, 2, false);
for ( size_t i = 0 ; i < my_lmdarr.length(0) ; i++ ) {
    sio.printf("my_lmdarr value[%zu]... [%lld]\n", i, my_lmdarr[i]);
}
```

Output:

```
my_lmdarr value[0]...  [-2147483646]
my_lmdarr value[1]...  [2147483647]
my_lmdarr value[2]...  [2147483647]
```

12.3.50 insert()

NAME

insert() — Insert an element

SYNOPSIS

```
mdarray_type &insert( ssize_t idx, size_t len );  ..... 1
mdarray_type &insert( size_t dim_index, ssize_t idx, size_t len );  ..... 2
```

DESCRIPTION

This member function inserts `len` values to the index `idx` in the array of the object itself. The values are default values.

For the member function 1, the insertion is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the insertion is applied to the elements in the dimension with the index `dim_index`.

PARAMETER

[I]	`idx`	Array subscript specifying an element before which new entries should be inserted
[I]	`len`	Number of elements to be inserted
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code inserts two default values (`0`) ahead of the first element of the object `my_mdarr` for he mdarray_long-class and prints the result to stdout for confirmation:

```
stdstreamio sio;

mdarray_long my_mdarr(false, 2);
my_mdarr[0] = -2147483646;
my_mdarr[1] = 2147483647;
my_mdarr.insert( 1, 2 );
for ( size_t i = 0 ; i < my_mdarr.length(0) ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%ld]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [-2147483646]
my_mdarr value[1]...  [0]
my_mdarr value[2]...  [0]
my_mdarr value[3]...  [2147483647]
```

12.3.51 crop()

NAME

crop() — Extract an element

SYNOPSIS

```
mdarray_type &crop( ssize_t idx, size_t len );  ..............................  1
mdarray_type &crop( size_t dim_index, ssize_t idx, size_t len );  .........  2
```

DESCRIPTION

This member function extracts `len` values from the index `idx` in the array of the object itself.

For the member function 1, the extraction is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the extraction is applied to the elements in the dimension with the index `dim_index`.

PARAMETER

[I]	`idx`	Array subscript specifying the first element to be extracted
[I]	`len`	Number of elements to be extracted
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code extracts the one-dimension elements specified by index 1 in the first dimension (dimension index: 0) from the mdarray_uchar-class object `my_cmdarr` and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_uchar my_cmdarr(false, 2, 3);
my_cmdarr(0,0) = 124;
my_cmdarr(1,0) = 125;
my_cmdarr(0,1) = 126;
my_cmdarr(1,1) = 127;
my_cmdarr.crop( 0, 1, 1 );
for ( size_t j = 0 ; j < my_cmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_cmdarr.length(0) ; i++ ) {
        sio.printf("my_cmdarr value(%zu, %zu)... [%hhu]\n",
                   i, j, my_cmdarr(i, j));
    }
}
```

Output:

```
my_cmdarr value(0, 0)...  [125]
my_cmdarr value(0, 1)...  [127]
my_cmdarr value(0, 2)...  [0]
```

12.3.52 erase()

NAME

erase() — Erase an element

SYNOPSIS

```
mdarray_type &erase( ssize_t idx, size_t len );  ..............................  1
mdarray_type &erase( size_t dim_index, ssize_t idx, size_t len );  ........  2
```

DESCRIPTION

This member function erases the specified elements from the array of the object itself. The length of the array is reduced by `len`.

For the member function 1, the erasing operation is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the erasing operation is applied to the elements in the dimension with the index `dim_index`.

PARAMETER

[I]	`idx`	Array subscript specifying the first element to be removed
[I]	`len`	Number of elements to be removed
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code erases the element specified by index 1 in the object `my_mdarr` with a one-dimension array and prints the values of elements to stdout:

```
stdstreamio sio;

mdarray_llong my_mdarr(false, 3);
my_mdarr[0] = 0;
my_mdarr[1] = 2147483646;
my_mdarr[2] = 2147483647;

my_mdarr.erase( 1, 1 );
for ( size_t i = 0 ; i < my_mdarr.length() ; i++ ) {
    sio.printf("my_mdarr value[%zu]... [%lld]\n", i, my_mdarr[i]);
}
```

Output:

```
my_mdarr value[0]...  [0]
my_mdarr value[1]...  [2147483647]
```

12.3.53 resize()

NAME

resize() — Change the length of the array

SYNOPSIS

```
mdarray_type &resize( size_t len );  ..... 1
mdarray_type &resize( size_t dim_index, size_t len );  ..... 2
mdarray_type &resize( const mdarray &src );  ..... 3
```

DESCRIPTION

This member function resizes the length of the array in the object itself.

For extension, new values of the elements are filled with the default values. For reduction, elements after the index `len` are removed.

For the member function 1, the resizing operation is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the resizing operation is applied to the elements in the dimension with the index `dim_index`. For the member function 3, the number of dimensions and the length of the array are conformed to that of the object `src`.

PARAMETER

[I]	`len`	Number of elements after being resized
[I]	`dim_index`	The integer number that specifies one of the dimensions of the array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code resizes the length of the array for the second dimension (dimension index: 1) in the object `my_cmdarr` with a two-dimension array to 3 and prints the result to stdout for confirmation:

```
stdstreamio sio;

mdarray_uchar my_cmdarr(false, 2, 2);
my_cmdarr(0,0) = 70;
my_cmdarr(1,0) = 71;
my_cmdarr(0,1) = 36;
my_cmdarr(1,1) = 37;

my_cmdarr.resize( 1, 3 );
for ( size_t j = 0 ; j < my_cmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_cmdarr.length(0) ; i++ ) {
        sio.printf("my_cmdarr value(%zu, %zu)... [%hhu]\n",
                    i, j, my_cmdarr(i, j));
    }
}
```

Output:

```
my_cmdarr value(0, 0)...  [70]
my_cmdarr value(1, 0)...  [71]
my_cmdarr value(0, 1)...  [36]
my_cmdarr value(1, 1)...  [37]
my_cmdarr value(0, 2)...  [0]
my_cmdarr value(1, 2)...  [0]
```

See §3.6.3 for an example of resizing the length of the array.

12.3.54 resizeby()

NAME

resizeby() — Change the length of the array relatively

SYNOPSIS

```
mdarray_type &resizeby( ssize_t len );  ..... 1
mdarray_type &resizeby( size_t dim_index, ssize_t len );  ..... 2
```

DESCRIPTION

This member function resizes the length of the array in the object itself by `len`.

For reduction, a negative value is passed to the argument `len`. The size of the resized array is the original size plus `len`.

For the member function 1, the resizing operation is always applied to the elements in the first dimension (dimension index: 0). For the member function 2, the resizing operation is applied to the elements in the dimension with the index `dim_index`.

PARAMETER

[I] `len` Number of elements to be increased or decreased
[I] `dim_index` The integer number that specifies one of the dimensions of the array
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer.

EXAMPLE

The following code resizes the length of the array in the object `my_cmdarr` with a one-dimension array and prints the values to stdout for confirmation:

```
stdstreamio sio;

mdarray_uchar my_cmdarr(false, 3);

my_cmdarr.resizeby( -2 );
for ( size_t i = 0 ; i < my_cmdarr.length(0) ; i++ ) {
    sio.printf("my_cmdarr value[%d]... [%hhu]\n", i, my_cmdarr[i]);
}
```

Output:

```
my_cmdarr value[0]...  [0]
```

12.3.55 increase_dim()

NAME
increase_dim() — Expand the number of dimensions

SYNOPSIS
```
mdarray_type &increase_dim();
```

DESCRIPTION
This member function increments the dimension of the array in the object itself.

RETURN VALUE
A reference to itself

EXCEPTION
The function throws an exception when it fails to allocate a local buffer.

EXAMPLE
The following code increments the dimension of the array in the object `my_mdarr` with a three-dimension array and prints the number of dimensions to stdout:

```
stdstreamio sio;

mdarray_uchar my_mdarr(false, 1, 2, 3);
my_mdarr.increase_dim();
sio.printf("my_mdarr dim... [%zu]\n",  my_mdarr.dim_length());
```

Output:
```
my_mdarr dim...  [4]
```

12.3.56 decrease_dim()

NAME
decrease_dim() — Reduce the number of dimensions

SYNOPSIS
```
mdarray_type &decrease_dim();
```

DESCRIPTION
This member function decrements the dimension of the array in the object itself.

RETURN VALUE
A reference to itself

EXCEPTION
The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE
The following code decrements the dimension of the array in the object `my_mdarr` with a three-dimension array and prints the number of dimensions to stdout:

```
stdstreamio sio;

mdarray_uchar my_mdarr(false, 1, 2, 3);
my_mdarr.decrease_dim();
sio.printf("my_mdarr dim... [%zu]\n",  my_mdarr.dim_length());
```

Output:

```
my_mdarr dim...  [2]
```

12.3.57 swap()

NAME

swap() — Replace the object by another one

SYNOPSIS

```
mdarray_type &swap( mdarray_type &sobj );
```

DESCRIPTION

This member function swaps the specified object `sobj` for the object itself. All attributes such as the size of arrays are exchanged.

PARAMETER

[I/O] `sobj` The object that belongs to the same class as this instance
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code swaps the object `my_fmdarr` with a one-dimension array for the object `swap_mdarr` and prints the values of elements in the `my_fmdarr` to stdout:

```
stdstreamio sio;

mdarray_float my_fmdarr(false, 2);
my_fmdarr[0] = 1000;
my_fmdarr[1] = 2000;

mdarray_float swap_mdarr(false, 2);
swap_mdarr[0] = 100;
swap_mdarr[1] = 200;
my_fmdarr.swap(swap_mdarr);
for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%g]\n", i, my_fmdarr.dvalue(i));
}
```

Output:

```
my_fmdarr value[0]...  [100]
my_fmdarr value[1]...  [200]
```

12.3.58 convert()

NAME

convert() — Convert the value of the full array element

SYNOPSIS

```
mdarray_type &convert( void (*func)(const type [],type [],size_t,bool,void *),
                       void *user_ptr );
```

DESCRIPTION

This member function converts the all values of the object itself via the user-defined function `func`.

The first argument of the user-defined function is the address of the original elements in the array; the second is the address of the elements that should be written by user's programs; the third is length of elements that should be converted in user-defined function, and the fifth is `user_ptr`. The fourth argument of the user-defined function should be ignored in user's programs.

PARAMETER

[I]	`func`	Address of user-defined function
[I]	`user_ptr`	The pointer that is given to the above function as its last argument

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

12.3.59 ceil()

NAME

ceil() — Raise decimals to the next whole number in a double type value

SYNOPSIS

```
mdarray_type &ceil();
```

DESCRIPTION

This member function rounds up all elements (floating-point number) of the array in the object itself to the nearest integer.

RETURN VALUE

A reference to itself

EXAMPLE

The following code sets the values with decimal places to the object `my_fmdarr` with a one-dimension array and rounds them up, and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr;

my_fmdarr[0] = 1000.1;
my_fmdarr[1] = 2000.6;
my_fmdarr.ceil();
```

```
for ( size_t i = 0 ; i < my_fmdarr.length() ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%f]\n", i, my_fmdarr[i]);
}
```

Output:

```
my_fmdarr value[0]...  [1001.000000]
my_fmdarr value[1]...  [2001.000000]
```

12.3.60 floor()

NAME

floor() — Devalue decimals in a double type value

SYNOPSIS

```
mdarray_type &floor();
```

DESCRIPTION

This member function rounds down all elements (floating-point number) of the array in the object itself to the nearest integer.

RETURN VALUE

A reference to itself

EXAMPLE

The following code sets the values with decimal places to the object `my_fmdarr` with a one-dimension array and rounds them down, and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr;

my_fmdarr[0] = 1000.1;
my_fmdarr[1] = 2000.9;
my_fmdarr.floor();

for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%f]\n", i, my_fmdarr[i]);
}
```

Output:

```
my_fmdarr value[0]...  [1000.000000]
my_fmdarr value[1]...  [2000.000000]
```

12.3.61 round()

NAME

round() — Round off decimals in a double type value

SYNOPSIS

```
mdarray_type &round();
```

DESCRIPTION

This member function rounds all elements (floating-point number) of the array in the object itself to the nearest integer.

RETURN VALUE

A reference to itself

EXAMPLE

The following code sets the values with decimal places to the object `mdarrf` with a one-dimension array and rounds them to the nearest integer, and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr;
my_fmdarr[0] = 1000.5;
my_fmdarr[1] = -1000.5;
my_fmdarr.round();

for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%f]\n", i, my_fmdarr[i]);
}
```

Output:

```
my_fmdarr value[0]...  [1001.000000]
my_fmdarr value[1]...  [-1001.000000]
```

12.3.62 trunc()

NAME

trunc() — Omit decimals in a double type value

SYNOPSIS

```
mdarray_type &trunc();
```

DESCRIPTION

This member function rounds down all elements (floating-point number) of the array in the object itself to the integer which is closer to 0 than the element.

RETURN VALUE

A reference to itself

EXAMPLE

The following code sets the values with decimal places to the object `my_fmdarr` with a one-dimension array and rounds them down to the integers which are closer to 0 than the elements, and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr;
my_fmdarr[0] = 1.7;
my_fmdarr[1] = -1.7;
```

```
my_fmdarr.trunc();

for ( size_t i = 0 ; i < my_fmdarr.length() ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%f]\n", i, my_fmdarr[i]);
}
```

Output:

```
my_fmdarr value[0]...  [1.000000]
my_fmdarr value[1]...  [-1.000000]
```

12.3.63 abs()

NAME

abs() — Absolute value of all elements

SYNOPSIS

```
mdarray_type &abs();
```

DESCRIPTION

This member function returns the absolute values of all elements of the array in the object itself.

RETURN VALUE

A reference to itself

EXAMPLE

The following code sets negative values to the elements of the object `my_fmdarr` with a one-dimension array and prints the absolute values of elements to stdout:

```
stdstreamio sio;

mdarray_float my_fmdarr;

my_fmdarr[0] = -1000.1;
my_fmdarr[1] = -2000.6;
my_fmdarr.abs();

for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
    sio.printf("my_fmdarr value[%zu]... [%5.1f]\n", i, my_fmdarr[i]);
}
```

Output:

```
my_fmdarr value[0]...  [1000.1]
my_fmdarr value[1]...  [2000.6]
```

12.3.64 compare()

NAME

compare() — Compare array objects

SYNOPSIS

```
bool compare(const mdarray &obj) const;
```

DESCRIPTION

This member function compares the array of the object itself with that of the specified object `obj`.

The argument is the mdarray class. This means that the object with the different type of the array from the object itself can be passed to it. Even when the data types are not identical, this member function returns true (=1) if the length and values of the array are identical. If not, it returns false (=0).

PARAMETER

[I] `obj` The object that belongs to a class derived from "mdarray"
([I] : input, [O] : output)

RETURN VALUE

true : If the sizes and values of the elements on the arrays are identical
false : If the sizes and one of the values of the elements on the arrays are not identical

EXAMPLE

The following code compares the object `my_fmdarr` with a two-dimension array with the object `my_i64mdarr` with a two-dimension array and prints the result to stdout:

```
stdstreamio sio;

mdarray_float my_fmdarr(false, 2,2);
my_fmdarr(0,0) = 1000;

mdarray_int64 my_i64mdarr(false, 2,2);
my_i64mdarr(0,0) = 1000;

sio.printf("*** my_fmdarr compare [%d] *** \n",
           (int)my_fmdarr.compare(my_i64mdarr));
```

Output:

```
*** my_fmdarr compare [1] ***
```

12.3.65 copy()

NAME

copy() — Copy an array into another object

SYNOPSIS

```
ssize_t copy( mdarray_type *dest ) const;
```

DESCRIPTION

This member function copies all the contents of the object itself to the specified object `dest`.

All attributes such as the length and values of the source array are copied to the destination array. This member function does not affect the array of the object itself (source).

PARAMETER

[O] `dest` The object to which this array is written
([I] : input, [O] : output)

RETURN VALUE

The number of copied elements (column × row × layer)

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code copies the object `my_cmdarr` with a two-dimension array to the object `my_fmdarr` and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_float my_fmdarr;
float my_values[] = {99, 101, 98, 102};
mdarray_float my_cmdarr(false, 2, 2, my_values);

ssize_t copy_size = my_cmdarr.copy( &my_fmdarr );

for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
        sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
                    i, j, my_fmdarr(i, j));
    }
}
```

Output:

```
my_fmdarr value(0,0)...  [98]
my_fmdarr value(1,0)...  [99]
my_fmdarr value(0,1)...  [101]
my_fmdarr value(1,1)...  [102]
```

12.3.66 copy()

NAME

copy() — Copy a part of an array into another object (for image data)

SYNOPSIS

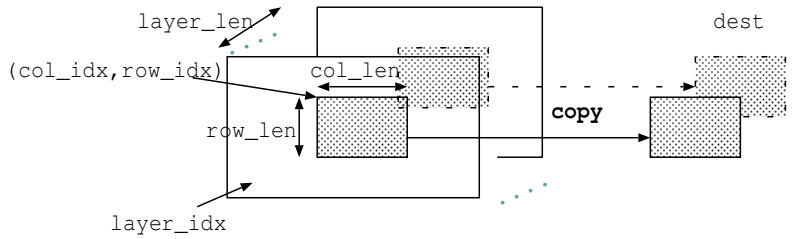
```
ssize_t copy( mdarray_type *dest,
              ssize_t col_idx, size_t col_len=MDARRAY_ALL,
              ssize_t row_idx=0, size_t row_len=MDARRAY_ALL,
              ssize_t layer_idx=0, size_t layer_len=MDARRAY_ALL ) const;
```

DESCRIPTION

This member function is for image data and copies a part of the contents of the object itself to the specified object `dest`.

This member function does not affect the array of the object itself (source).

Image data is copied by copy() as shown below:



The shaded area in the figure is specified by the second or later arguments, and it is copied to the `dest`.

Do not specify `MDARRAY_ALL` for an argument explicitly.

PARAMETER

[O]	`dest`	An instance of the class "mdarray_type" to which a subarray of this object should be written
[I]	`col_idx`	Subscript specifying the first column of the subarray
[I]	`col_len`	Number of columns in the subarray
[I]	`row_idx`	Subscript specifying the first row of the subarray
[I]	`row_len`	Number of rows in the subarray
[I]	`layer_idx`	Subscript specifying the first layer of the subarray
[I]	`layer_len`	Number of layers in the subarray

([I] : input, [O] : output)

RETURN VALUE

The number of copied elements (column × row × layer)

EXCEPTION

The function throws an exception when it fails to allocate a buffer, or it detects memory corruption.

EXAMPLE

The following code copies the object `my_cmdarr` with a two-dimension array to the object `my_fmdarr` and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

double my_vals[] = {98, 99, 101, 102};
mdarray_double my_cmdarr(false, 2,2, my_vals);

double my_d[] = {-501, 501, -502, 502};
mdarray_double my_dmdarr(false, 2,2, my_d);

ssize_t ret_size = my_cmdarr.copy( &my_dmdarr, 1, 1, 1, 1 );
for ( size_t j = 0 ; j < my_dmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_dmdarr.length(0) ; i++ ) {
        sio.printf("my_dmdarr value(%zu,%zu)... [%g]\n",
                    i, j, my_dmdarr(i, j));
    }
}
```

Output:

```
my_ldmdarr value(0,0)...  [102]
```

See §3.6.7 for an example of copy and paste.

12.3.67 cut()

NAME

cut() — Cut all values in an array and copy them into another object

SYNOPSIS

```
mdarray_type &cut( mdarray_type *dest );
```

DESCRIPTION

This member function cuts all contents of the array of the object itself and copies it to the specified object `dest`.

Since all contents of the array of the object itself are cut, the length of the array (source) is set to 0.

PARAMETER

[O] `dest` The object to which this array is written
([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a buffer, or it detects memory corruption.

EXAMPLE

The following code cuts the object `my_cmdarr` with a two-dimension array and copies it to the object `my_mdarr`, and prints the length of the array of the `my_cmdarr` to stdout for confirmation:

```
stdstreamio sio;

mdarray_uchar my_mdarr;

unsigned char my_char[] = {51, 101, 52, 102};
mdarray_uchar my_cmdarr(false, 2, 2, my_char);

my_cmdarr.cut( &my_mdarr );
sio.printf("my_cmdarr length()... [%zu]\n", my_cmdarr.length());
```

Output:

```
my_cmdarr length()...  [0]
```

12.3.68 cut()

NAME

cut() — Cut a part of values in an array and copy them into another object (for image data)

SYNOPSIS

```
mdarray_type &cut( mdarray_type *dest,
                   ssize_t col_idx, size_t col_len=MDARRAY_ALL,
```

```
                    ssize_t row_idx=0, size_t row_len=MDARRAY_ALL,
                    ssize_t layer_idx=0, size_t layer_len=MDARRAY_ALL );
```

DESCRIPTION

This member function is for image data and cuts a part of the contents of the object itself, and copies it to the specified object `dest`.

The length of the array of the object itself (source) is not changed and the values of the area specified by the second or later arguments are filled with default values.

Do not specify `MDARRAY_ALL` for an argument explicitly.

PARAMETER

[O]	`dest`	An instance of the class "mdarray_type" to which a subarray of this object should be written
[I]	`col_idx`	Subscript specifying the first column of the subarray
[I]	`col_len`	Number of columns in the subarray
[I]	`row_idx`	Subscript specifying the first row of the subarray
[I]	`row_len`	Number of rows in the subarray
[I]	`layer_idx`	Subscript specifying the first layer of the subarray
[I]	`layer_len`	Number of layers in the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXCEPTION

The function throws an exception when it fails to allocate a local buffer, or it detects memory corruption.

EXAMPLE

The following code cuts the zeroth column of the object `my_cmdarr` with a two-dimension array and copies it to the object `my_mdarr` and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_uchar my_mdarr;

unsigned char my_char[] = {51, 101, 52, 102};
mdarray_uchar my_cmdarr(false, 2,2, my_char);

my_cmdarr.cut( &my_mdarr, 0, 1 );
for ( size_t j = 0 ; j < my_cmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_cmdarr.length(0) ; i++ ) {
        sio.printf("my_cmdarr value(%zu,%zu)... [%hhu]\n",
                    i, j, my_cmdarr(i, j));
    }
}
```

Output:

```
my_cmdarr value(0,0)...  [0]
my_cmdarr value(1,0)...  [101]
my_cmdarr value(0,1)...  [0]
my_cmdarr value(1,1)...  [102]
```

12.3.69 clean()

NAME

clean() — Padding of existing values in an array by default ones (for image data)

SYNOPSIS

```
mdarray_type &clean( ssize_t col_index = 0, size_t col_size = MDARRAY_ALL,
                     ssize_t row_index = 0, size_t row_size = MDARRAY_ALL,
                     ssize_t layer_index = 0, size_t layer_size = MDARRAY_ALL );
```

DESCRIPTION

This member function fills the array elements of the object itself with default values. The arguments are optional. When no argument is specified, the cleaning operation is applied to all elements. The length of the array is not changed by `clean()`.

This member function is for image data.

Do not specify `MDARRAY_ALL` for an argument explicitly.

PARAMETER

[I]	`col_index`	Subscript specifying the first column of a subarray in this object
[I]	`col_size`	Number of columns of the subarray
[I]	`row_index`	Subscript specifying the first row of the subarray
[I]	`row_size`	Number of rows of the subarray
[I]	`layer_index`	Subscript specifying the first layer of the subarray
[I]	`layer_size`	Number of layers of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code sets values to the elements of the object `my_smdarr` with a two-dimension array and cleans an element, and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_short my_smdarr(false, 2,2);
my_smdarr(0,0) = 1;
my_smdarr(1,0) = 3;
my_smdarr(0,1) = 2;
my_smdarr(1,1) = 4;

my_smdarr.clean(1,1,1,1);
for ( size_t j = 0 ; j < my_smdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_smdarr.length(0) ; i++ ) {
        sio.printf("my_smdarr value(%zu,%zu)... [%hd]\n",
                   i, j, my_smdarr(i, j));
    }
}
```

Output:

```
my_smdarr value(0,0)...  [1]
```

```
my_smdarr value(1,0)...  [3]
my_smdarr value(0,1)...  [2]
my_smdarr value(1,1)...  [0]
```

12.3.70 fill()

NAME

fill() — Rewrite element values (for image data)

SYNOPSIS

```
mdarray_type &fill( double value,
    ssize_t col_index = 0, size_t col_size = MDARRAY_ALL,
    ssize_t row_index = 0, size_t row_size = MDARRAY_ALL,
    ssize_t layer_index = 0, size_t layer_size = MDARRAY_ALL );  ..... 1
mdarray_type &fill( double value,
    void (*func)(double [],double,size_t,
                 ssize_t,ssize_t,ssize_t,mdarray_type *,void *),
    void *user_ptr,
    ssize_t col_index = 0, size_t col_size = MDARRAY_ALL,
    ssize_t row_index = 0, size_t row_size = MDARRAY_ALL,
    ssize_t layer_index = 0, size_t layer_size = MDARRAY_ALL );  ..... 2
```

DESCRIPTION

This member function rewrites the specified array elements of the object itself as the argument `value` (Function 1). This member function rewrites the specified array elements of the object itself via the user-defined function `func` (Function 2).

The arguments of the user-defined function are, from the left, the values of double type stored in temporary buffer converted from that in the object itself, the value specified by `value`, length of elements in temporary buffer that should be modified by user's programs, the index of a column, the index of a row, the index of a layer, the address of the object itself, and the user pointer `user_ptr`, respectively. User's programs should modify elements in temporary buffer. To find out how to specify a user-defined function, see EXAMPLE in §12.3.73.

This member function is for image data.

Do not specify `MDARRAY_ALL` for an argument explicitly.

PARAMETER

[I]	`value`	A real scalar to be written to a subarray of this object
[I]	`user_ptr`	The pointer that is given to user-defined function as its last argument
[I]	`col_index`	Subscript specifying the first column of the subarray
[I]	`col_size`	Number of columns of the subarray
[I]	`row_index`	Subscript specifying the first row of the subarray
[I]	`row_size`	Number of rows of the subarray
[I]	`layer_index`	Subscript specifying the first layer of the subarray
[I]	`layer_size`	Number of layers of the subarray
[I]	`func`	The pointer to user-defined function that defines an operation to be performed on each element of this array

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code fills all elements of the object `my_smdarr` with a two-dimension array with `100` and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

mdarray_short my_smdarr(false, 2,2);
my_smdarr.fill(100);

for ( size_t j = 0 ; j < my_smdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_smdarr.length(0) ; i++ ) {
        sio.printf("my_smdarr value(%zu,%zu).. [%hd]\n",
                   i, j, my_smdarr(i, j));
    }
}
```

Output:

```
my_smdarr value(0,0)..  [100]
my_smdarr value(1,0)..  [100]
my_smdarr value(0,1)..  [100]
my_smdarr value(1,1)..  [100]
```

12.3.71 add()

NAME

add() — Add element values (for image data)

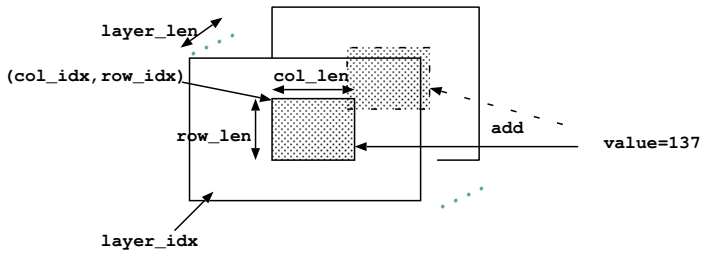
SYNOPSIS

```
mdarray_type &add( double value,
                   ssize_t col_index = 0, size_t col_size = MDARRAY_ALL,
                   ssize_t row_index = 0, size_t row_size = MDARRAY_ALL,
                   ssize_t layer_index = 0, size_t layer_size = MDARRAY_ALL );
```

DESCRIPTION

This member function adds the value of the argument `value` to array elements of the object itself specified by arguments. This member function is for image data.

The value 137 is added to a part of image data by add() as shown below:



The shaded area in the figure is specified by the second or later arguments, and `value` is added to the area.

Do not specify `MDARRAY_ALL` for an argument explicitly.

PARAMETER

[I]	`value`	A real scalar to be added to a subarray of this object
[I]	`col_index`	Subscript specifying the first column of the subarray
[I]	`col_size`	Number of columns of the subarray
[I]	`row_index`	Subscript specifying the first row of the subarray
[I]	`row_size`	Number of rows of the subarray
[I]	`layer_index`	Subscript specifying the first layer of the subarray
[I]	`layer_size`	Number of layers of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code adds `10` to the value in the second column and second row of the object `my_smdarr` with a two-dimension array and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

short my_short[] = {1, 2, 3, 4};
mdarray_short my_smdarr(false, 2,2, my_short);

my_smdarr.add(10.0, 1,1,1,1);

for ( size_t j = 0 ; j < my_smdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_smdarr.length(0) ; i++ ) {
        sio.printf("my_smdarr value(%zu,%zu)... [%hd]\n",
                   i, j, my_smdarr(i, j));
    }
}
```

Output:

```
my_smdarr value(0,0)...  [1]
my_smdarr value(1,0)...  [2]
my_smdarr value(0,1)...  [3]
my_smdarr value(1,1)...  [14]
```

12.3.72 multiply()

NAME

multiply() — Multiply element values (for image data)

SYNOPSIS

```
mdarray_type &multiply( double value,
                        ssize_t col_index = 0, size_t col_size = MDARRAY_ALL,
                        ssize_t row_index = 0, size_t row_size = MDARRAY_ALL,
                        ssize_t layer_index = 0, size_t layer_size = MDARRAY_ALL );
```

DESCRIPTION

This member function multiplies the specified array elements of the object itself by the value of the argument `value`. This member function is for image data.

Do not specify MDARRAY_ALL for an argument explicitly.

PARAMETER

[I]	value	A real scalar to be multiplied to a subarray of this object
[I]	col_index	Subscript specifying the first column of the subarray
[I]	col_size	Number of columns of the subarray
[I]	row_index	Subscript specifying the first row of the subarray
[I]	row_size	Number of rows of the subarray
[I]	layer_index	Subscript specifying the first layer of the subarray
[I]	layer_size	Number of layers of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code multiplies all elements of the object my_fmdarr with a two-dimension array by 50 and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

float my_float[] = {1, 3, 2, 4};
mdarray_float my_fmdarr(false, 2,2, my_float);

my_fmdarr.multiply(50);
for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
        sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
                   i, j, my_fmdarr(i, j));
    }
}
```

Output:

```
my_fmdarr value(0,0)...  [50.000000]
my_fmdarr value(1,0)...  [150.000000]
my_fmdarr value(0,1)...  [100.000000]
my_fmdarr value(1,1)...  [200.000000]
```

12.3.73 paste()

NAME

paste() — Paste up an array object (for image data)

SYNOPSIS

```
mdarray_type &paste( const mdarray &src,
    ssize_t dest_col = 0, ssize_t dest_row = 0, ssize_t dest_layer = 0 );   1
mdarray_type &paste( const mdarray &src,
    void (*func)(double [],double [],size_t,
                 ssize_t,ssize_t,ssize_t,mdarray_type *,void *),
    void *user_ptr,
    ssize_t dest_col = 0, ssize_t dest_row = 0, ssize_t dest_layer = 0 );   2
```

DESCRIPTION

This member function pastes the element values specified by `src` into the specified region of the array in the object itself (Function 1). This member function pastes the element values converted via the user-defined function into the specified region of the array in the object itself (Function 2).

The first argument is the mdarray class. This means that the object with the different type of the array from the object itself can be passed to it.

For the member function 2, the behavior of pasting is customizable by a user-defined function. The arguments of the user-defined function `func` are, from the left, the values of double type stored in temporary buffer converted from that in the object itself, the values of double type stored in temporary buffer converted from that in object `src`, length of elements in temporary buffer that should be modified by user's programs, the index of a column, the index of a row, the index of a layer, the address of the object itself, and the user pointer `user_ptr`, respectively. User's programs should modify elements in temporary buffer pointed by first argument. This member function is for image data.

PARAMETER

[I]	`src`	An instance of the class "mdarray" containing an input array
[I]	`func`	A pointer to user-defined function for computing a new scalar to be pasted
[I]	`user_ptr`	The pointer that is given to the user-defined function "func" as its last argument
[I]	`dest_col`	Subscript specifying the first column of a subarray of this object on which the input array is pasted
[I]	`dest_row`	Subscript specifying the first row of the subarray
[I]	`dest_layer`	Subscript specifying the first layer of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code pastes the object `mypaste_mdarr` with a two-dimension array into the object `my_fmdarr` with a two-dimension array. In pasting, the values of the objects are summed up and 500 is added to the values via the user-defined function. The values of elements are printed to stdout for confirmation:

```
void my_func(double self[], double src[], size_t len, ssize_t x,
             ssize_t y, ssize_t z, mdarray_float *myptr, void *p)
{
    size_t i;
    for ( i=0 ; i < len ; i++ ) self[i] += src[i] + 500;
}

/* main */
stdstreamio sio;

float my_float[] = {100, 0, 200, 0};
mdarray_float my_fmdarr(false, 2,2, my_float);

float mypaste_float[] = {1000, 3000, 2000, 4000};
```

```
mdarray_float mypaste_mdarr(false, 2,2, mypaste_float);

my_fmdarr.paste(mypaste_mdarr, &my_func, NULL);
for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
        sio.printf("my_fmdarr value[%zu][%zu]... [%f]\n", i, j,
                   my_fmdarr(i, j));
    }
}
```

Output:

```
my_fmdarr value(0,0)...  [1600.000000]
my_fmdarr value(1,0)...  [2500.000000]
my_fmdarr value(0,1)...  [3700.000000]
my_fmdarr value(1,1)...  [4500.000000]
```

See §3.6.7 for an example of copy and paste.

12.3.74 add()

NAME

add() — Add an array object (for image data)

SYNOPSIS

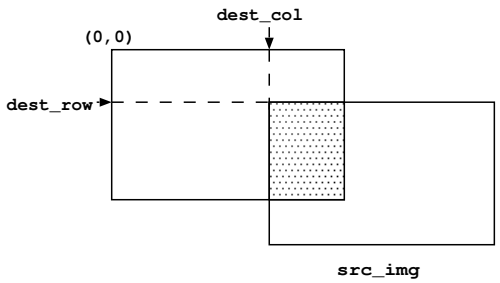
```
mdarray_type &add( const mdarray &src_img, ssize_t dest_col = 0,
                   ssize_t dest_row = 0, ssize_t dest_layer = 0 );
```

DESCRIPTION

This member function adds array elements of the object `src_img` to those of the object itself. A start position for addition can be specified separately for columns, rows, and layers. This member function is for image data.

The first argument is the mdarray class. This means that the object with the different type of the array from the object itself can be passed to it.

Image data is added to the elements of the object itself by add() as shown below:



The shaded area in the figure is specified by the second or later arguments, and `src_img` is added to the area.

PARAMETER

[I]	`src_img`	An instance of the class "mdarray" to be added to a subarray of this object
[I]	`dest_col`	Subscript specifying the first column of the subarray
[I]	`dest_row`	Subscript specifying the first row of the subarray
[I]	`dest_layer`	Subscript specifying the first layer of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code adds the object `add_smdarr` with a two-dimension array to the object `my_smdarr` with a two-dimension array and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

short my_short[] = {1, 2, 3, 4};
mdarray_short my_smdarr(false, 2,2, my_short);

short myadd_short[] = {9, 8, 7, 6};
mdarray_short myadd_smdarr(false, 2,2, myadd_short);

my_smdarr.add(myadd_smdarr);
for ( size_t j = 0 ; j < my_smdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_smdarr.length(0) ; i++ ) {
        sio.printf("my_smdarr value(%zu,%zu)... [%hd]\n",
                   i, j, my_smdarr(i, j));
    }
}
```

Output:

```
my_smdarr value(0,0)...  [10]
my_smdarr value(1,0)...  [10]
my_smdarr value(0,1)...  [10]
my_smdarr value(1,1)...  [10]
```

12.3.75 subtract()

NAME

subtract() — Subtract an array object (for image data)

SYNOPSIS

```
mdarray_type &subtract( const mdarray &src_img, ssize_t dest_col = 0,
                        ssize_t dest_row = 0, ssize_t dest_layer = 0 );
```

DESCRIPTION

This member function subtracts array elements of the object `src_img` from those of the object itself. A start position for subtraction can be specified separately for columns, rows, and layers. This member function is for image data.

The first argument is the mdarray class. This means that the object with the different type of the array from the object itself can be passed to it.

PARAMETER

[I]	`src_img`	An instance of the class "mdarray" to be subtracted from a subarray of this object
[I]	`dest_col`	Subscript specifying the first column of the subarray
[I]	`dest_row`	Subscript specifying the first row of the subarray
[I]	`dest_layer`	Subscript specifying the first layer of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code subtracts the object `mysubtract_mdarr` with a two-dimension array from the object `my_fmdarr` with a two-dimension array and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

float my_float[] = {1000, 2000, 3000, 4000};
mdarray_float my_fmdarr(false, 2,2, my_float);

float mysubt_float[] = {100, 200, 300, 400};
mdarray_float mysubtract_mdarr(false, 2,2, mysubt_float);

my_fmdarr.subtract(mysubtract_mdarr);
for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
        sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
                   i, j, my_fmdarr(i, j));
    }
}
```

Output:

```
my_fmdarr value(0,0)...  [900.000000]
my_fmdarr value(1,0)...  [1800.000000]
my_fmdarr value(0,1)...  [2700.000000]
my_fmdarr value(1,1)...  [3600.000000]
```

12.3.76 multiply()

NAME

multiply() — Multiply an array object (for image data)

SYNOPSIS

```
mdarray_type &multiply( const mdarray &src_img, ssize_t dest_col = 0,
                        ssize_t dest_row = 0, ssize_t dest_layer = 0 );
```

DESCRIPTION

This member function multiplies array elements of the object itself by those of the object `src_img`. A start position for multiplication can be specified separately for columns, rows, and layers. This member function is for image data.

The first argument is the mdarray class. This means that the object with the different type of the array from the object itself can be passed to it.

PARAMETER

[I]	`src_img`	An instance of the class "mdarray" by which a subarray of this object is multiplied
[I]	`dest_col`	Subscript specifying the first column of the subarray
[I]	`dest_row`	Subscript specifying the first row of the subarray
[I]	`dest_layer`	Subscript specifying the first layer of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code multiplies the object `my_fmdarr` with a two-dimension array by the object `mymulti_fmdarr` with a two-dimension array and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

float my_float[] = {1, 2, 3, 4};
mdarray_float my_fmdarr(false, 2,2, my_float);

float mymulti_float[] = {10, 20, 30, 40};
mdarray_float mymulti_fmdarr(false, 2,2, mymulti_float);

my_fmdarr.multiply(mymulti_fmdarr);
for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
        sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
                   i, j, my_fmdarr(i, j));
    }
}
```

Output:

```
my_fmdarr value(0,0)...  [10.000000]
my_fmdarr value(1,0)...  [40.000000]
my_fmdarr value(0,1)...  [90.000000]
my_fmdarr value(1,1)...  [160.000000]
```

12.3.77 divide()

NAME

divide() — Divide an array object (for image data)

SYNOPSIS

```
mdarray_type &divide( const mdarray &src_img, ssize_t dest_col = 0,
                      ssize_t dest_row = 0, ssize_t dest_layer = 0 );
```

DESCRIPTION

This member function divides array elements of the object itself by those of the object `src_img`. A start position for division can be specified separately for columns, rows, and layers. This member function is for image data.

The first argument is the mdarray class. This means that the object with the different type of the array from the object itself can be passed to it.

PARAMETER

[I]	`src_img`	An instance of the class "mdarray" by which a subarray of this object is divided
[I]	`dest_col`	Subscript for the first column of the subarray
[I]	`dest_row`	Subscript for the first row of the subarray
[I]	`dest_layer`	Subscript for the first layer of the subarray

([I] : input, [O] : output)

RETURN VALUE

A reference to itself

EXAMPLE

The following code divides the object `my_fmdarr` with a two-dimension array by the object `mydiv_mdarrf` with a two-dimension array and prints the values of elements to stdout for confirmation:

```
stdstreamio sio;

float my_float[] = {1000, 2000, 3000, 4000};
mdarray_float my_fmdarr(false, 2,2, my_float);

float mydiv_float[] = {2, 4, 6, 8};
mdarray_float mydiv_mdarrf(false, 2,2, mydiv_float);

my_fmdarr.divide(mydiv_mdarrf);
for ( size_t j = 0 ; j < my_fmdarr.length(1) ; j++ ) {
    for ( size_t i = 0 ; i < my_fmdarr.length(0) ; i++ ) {
        sio.printf("my_fmdarr value(%zu,%zu)... [%f]\n",
        i, j, my_fmdarr(i, j));
    }
}
```

Output:

```
my_fmdarr value(0,0)...  [500.000000]
my_fmdarr value(1,0)...  [500.000000]
my_fmdarr value(0,1)...  [500.000000]
my_fmdarr value(1,1)...  [500.000000]
```

Changes in APIs from the version 1.0 series

Member functions name changes

- `tstring::substr()` → `tstring::crop()`

  The substr() member function replaces a string of its own with other parts of a string of its own. Its name was changed to `crop()` because it has different behavior to the substr() function that is as generally used. It operates in the same manner as `substr()` formerly did.

  The copy() member function is used the same as the substr() function is generally. `copy()`enables parts of its own string to be copied to external objects.

- `tstring::strltrim()` → `tstring::ltrim()`
  `tstring::strrtrim()` → `tstring::rtrim()`
  `tstring::strtrim()` → `tstring::trim()`

  The name of these member functions were changed because `strrtrim()` was too long and difficult to read.

  `tstring::strtrim()` can still be used but use of `tstring::trim()` is advisable in the future.

Member functions with argument changes

- `tstring::strtol()`, `tstring::strtoll()`, `tstring::strtoul()`, `tstring::strtoull()`

  Changes were made to ensure that the argument "`size_t *endpos`" always comes at the end. The argument "`int base`" is replaced with the argument "`size_t *endpos`".

  Compiling codes without having changed them will result in an error being reported.

- `tarray_tstring::split()`,
  `asarray_tstring::split_keys()`,
  `asarray_tstring::split_values()`

  Only the argument of "`bool rm_escape`" has been added to the last. Specifying this new argument allows whether or not to delete escape characters in strings after a division to be specified. For example, if the argument is `true` the original string "`program\ files`" becomes "`program files`" after the division. However, with any parts parenthesized by quotations the escape characters are not deleted, irrespective of `rm_escape`.

  To ensure the same operation as in the version 1.0 series set `true` to `rm_escape`.

  Compiling codes without having changing them will result in an error being reported.

  The argument `delims` can use expressions like "`"[A-Z]"`". For more details on the expressions refer to tstring::trim() (§9.5.26).

Member functions which use of other APIs should be considered, depending on the situation

- `tstring::regmatch( const char *pat )`,
  `tstring::regmatch( size_t pos, const char *pat )`

  `tarray_tstring::regassign()` is more useful when you need to retrieve back reference information and further process the information.

  `tarray_tstring::regassign()` attempts matching on the string provided by an argument, and stores any parts that match a regular expression and substrings that are back-referenced by those parts as a string array.

Member functions that have been enhanced by overloads

- The const version was added to the `[]` operators and `at()` member function for each class.
- `tstring::find()`, `tstring::strpbrk(), tstring::regmatch()` etc can be provided with a pointer argument to acquire the next search position.

Primary member functions that have been added

- `strreplace()`, `chomp()`, `trim()`, `tolower()`, `toupper()`, `regreplace()` etc were added to the tarray_tstring class and asarray_tstring class, and all the elements of an array to be changed at once.
- Search APIs such as `find()`, `find_elem()`, `regmatch()` were added to the tarray_tstring class.

  The asarray_tstring also makes these search APIs available for use through the values() member function and keys() member function.