Parallel C++
User Guide

3L Ltd

This edition November 22, 1991 describes version 2.1 of the software.

20 19 18 17 16 15 14 13 12 11
10 9 8 7 6 5 4 3 2 1

3L Ltd
Peel House
Ladywell
Livingston EH54 6AG
Scotland
Tel. +44 506 41 59 59
Fax. +44 506 41 59 44
E-mail Support@ThreeL.Co.UK

Contents

Introduction

Intended Audience

This *User Guide* accompanies 3L's Parallel C++ product, and is intended for anyone who wants to use Parallel C++ to program a transputer system, whether writing a conventional sequential program or using the full support for concurrency which the transputer processor has to offer.

Parallel C++ is a "sister" of 3L Parallel C, and this manual should be read in conjuction with the *User Guide*[6] for that product.

Hardware Assumptions

Parallel C++ can be used with a large variety of target transputer systems. This manual makes the simplifying assumption that the target hardware will be an Inmos IMS B004 transputer evaluation board, or a transputer system which is largely compatible with a B004. This board is a single plug-in card for the standard IBM PC bus, with one transputer and either 1MB or 2MB of RAM.

Similarly, the assumption is made here that the host computer for the B004 will be an IBM PC with a hard disk drive, or one of the many personal computers compatible with the original IBM machines.

Document Structure

There are four main divisions within this document, as follows:

- *Part I: Getting Started* covers installing Parallel C++ on your machine and verifying that it is operating correctly.
- *Part II: Tutorial* introduces you to the operation of the compiler and the other tools supplied with Parallel C++.
- *Part III: Reference* contains detailed technical information about the compiler and Parallel C++ class libraries.
- The appendices at the end of this manual contain supplementary information in a condensed form.

Further Reading

This *User Guide* does not attempt to teach the C++ language itself. Instead, we suggest that the reader should consult one of the many introductory texts now available, such, for example, *Teach Yourself C++*[3], by Al Stevens, or *Programming in C++*[4], by Stephen C. Dewhurst and Kathy T. Stark. The classic description of C++ is, of course, Bjarne Stroustrup's *The C++ Programming Language*[1], although the language has changed significantly since its publication. A thorough description of the language as it now stands can be found in *The Annotated C++ Reference Manual*[2] by Margaret A. Ellis and Bjarne Stroustrup.

The reader is also assumed to be reasonably familiar with the operating system of the host computer being used. For personal computers made by IBM, this will usually be PC-DOS, which is supplied with a manual called *Disk Operating System Reference*[7]. For compatible machines made by other manufacturers, the operating system will usually be MS-DOS, described in *Microsoft MS-DOS User's Reference*[8]. These two operating systems are largely compatible, and their documentation is very similar. We will refer to "MS-DOS" in this manual to mean the operating system used on your machine. The term *DOS Reference Manual* will be used to refer to the appropriate manual.

References to these and other documents mentioned in this manual are collected in a bibliography, which can be found on page 147.

Conventions

Throughout this manual, text `printed in this typeface` represents direct verbatim communication with the computer: for example, pieces of C++ text, commands to MS-DOS and responses from the computer.

In examples, text *printed in this typeface* is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of one kind of compilation command:

`t8cc` *source-file*

This means that the command consists of:

1. The word "`t8cc`", typed exactly like that.

2. A *source-file*: not the text `source-file`, but an item of the *source-file* class, for example "`myprog.cpp`".

Part I

Getting Started

Chapter 1

Installing the Compiler

This chapter contains instructions on how to load Parallel C++ from the supplied floppy disks onto a hard disk and make it ready for use.

Parallel C++ must be installed in the same directory as your Parallel C kit. The current version of Parallel C++ must be installed in directory `\tc2v2`. This means that, for the present, your Parallel C kit must be installed in `\tc2v2` if Parallel C++ is to work correctly. You can find instructions for installing Parallel C in chapter 1 of the Parallel C *User Guide*[6].

You can skip this chapter if Parallel C++ has already been installed on the machine you are using.

1.1 Installing the Software

The compiler is distributed on two 360KB floppy disks. The contents of these disks are listed in appendix A.

To install Parallel C++ on your hard disk, follow this procedure.

1. Place the disk labelled `Disk 1 of 2` in your floppy disk drive `A:`.

2. Type the following commands:

```
C>a:

A>install
```

3. Answer any questions the `install` program asks you.

4. Place the appropriate disks in drive `A:` when the `install` program asks for them.

It is important to use the supplied `install` program to install Parallel C++. If you simply copy the files, the installation will not be performed correctly.

1.2 The Search Path

The compiler is now installed, but can only be run in the installation directory, `\tc2v2`. Before the compiler can be used from other directories the installation directory must be added to the MS-DOS *search path*. Program files stored in directories which are on the search path can be loaded and run simply by typing the name of the program as a command. So, to make sure that the C compiler is available as a command (`t8cc` or `t4cc`), the installation directory must be added to the search path.

The search path for your machine is set up by the batch file `c:\autoexec.bat` which is automatically executed when the machine starts up. To change the path, you will need to edit the `autoexec.bat` file using a text editor like `edlin`. (The *DOS Reference Manual* explains how to use `edlin`). Your `autoexec.bat` file will probably already contain a line of the following form:

`path` ... *list of directories* ...

For example:

```
path c:\dos;c:\utils
```

In this case, you will need to add the text "`c:\tc2v2`" on to the end of the line, giving:

```
path c:\dos;c:\utils;c:\tc2v2
```

If there is no `path` line in the `autoexec.bat` file, just add the line:

```
path c:\tc2v2
```

Some important points about setting the search path should be noted:

1. If you are a user of the Inmos TDS environment, your search path will probably include a reference to the directory where the TDS is held, such as `\tds2dir`. This reference must not precede the Parallel C++ installation directory in the path; if it does, the wrong version of the `afserver` program will be called.

2. From time to time, 3L release new versions of components, such as the linker or the `afserver`, which are included in more than one compiler product. This means that if you are a user of any other 3L compilers, you should make sure that the installation directory of the latest compiler product precedes all the others. This will ensure that the latest versions of these common components are picked up; they will be compatible with all the compiler products.

Once your `autoexec.bat` file has been changed, you will need to reboot your machine to make the changes effective.

Chapter 2

Confidence Testing

This chapter describes a short procedure which may be followed to check that installation has been done correctly.

1. Set the current disk drive to the one on which Parallel C++ has been installed. For example, if the compiler has been installed in directory `c:\tc2v2`, do this:

   ```
   D>c:

   C>
   ```

2. Set the current directory to a convenient directory for doing this test. For example:

   ```
   C>cd \mine

   C>
   ```

   NB: Don't use the installation directory for the confidence test, as this would mean that you would not be testing whether the correct search path has been set up.

3. Copy the example `hello.cpp` file to the current directory. If the installation directory is `\tc2v2`, for example, you should type this:

```
C>copy \tc2v2\examples\hello.cpp
     1 File(s) copied

C>
```

4. Compile the example using the T8 version of the compiler (this will work for the T4 as well, because the example contains no floating-point instructions). At the same time, we can check that the correct version of the compiler is available, by typing the following command. You should see the output shown.

```
C>t8cc /i hello
Transputer C++ compiler V2.1.1
Copyright (C) 3L Limited 1991
Portions Copyright (C) Computer Innovations, Inc. 1991
Portions Copyright (C) AT&T 1990

C>
```

If the above message does not appear, check the installation procedure, and in particular, ensure that the correct `path` command has been set up.

If instead the computer outputs the following, or something similar, it is likely that there has been some error in setting up the transputer board.

```
Last command = 0
Server terminated: bad protocol when expecting INT32
```

If this happens, please check in particular that the wire links, accessible from the back of the PC, have been correctly installed. The transputer board's documentation should help with this.

5. Link the resulting binary file with the necessary parts of theParallel C++ and C run-time libraries, and the harness:

```
C>t8cclink hello

C>linkt     hello C:\tc2v2\libct2 C:\tc2v2\crtlt8
C:\tc2v2\t8harn

C>
```

6. Finally, the program can be run:

```
C>afserver -:b hello.b4
Hello, world.

C>
```

The output "`Hello, world.`" comes from the `hello.cpp` example program. If it does not appear, we recommend that the installation procedure should be carefully repeated, and the confidence test procedure followed again. If this message still does not appear, please contact your dealer for further assistance.

Part II

Tutorial

Chapter 3

Developing C++ Programs

This chapter shows you how to build C++ programs to run on the transputer. Sections 3.1 to 3.4 discuss the use of the compiler and linker to produce conventional sequential programs. Section 3.5 deals with parallel programming in C++.

The instructions in this chapter assume that the Parallel C++ has already been installed as described in chapter 1.

Some of the procedures described here are different for T4 and T8 transputers. You should find out which type of transputer is fitted in your PC before using the compiler.

3.1 Compiling

Parallel C++ source programs are held in standard MS-DOS text files. These can be created any of the usual text editors.

A source program is compiled into a binary object (`.bin`) file of T8 transputer instructions by a command of the form:

```
t8cc source-file
```

To compile code for a T4 transputer, use the command

```
t4cc source-file
```

Note that, in general, code compiled for a T4 will not run on a T8 (or vice versa) so you must use the command appropriate for the type of processor on your transputer board.

The *source-file* is the filename of the C++ source program which is to be compiled. If no filename extension is given in the command, `.cpp` is added automatically.

So, to compile the file `hello.cpp` for the T8, you would give the command:

```
C>t8cc hello
```

If the source file contains no errors, an output object file `hello.bin` is produced. If the compiler detects errors in the source program, it writes diagnostic messages to the MS-DOS standard output stream.

3.2 Linking

Once a Parallel C++ program has been compiled into an object (`.bin`) file, it must be linked with any external functions it requires before it can be run, including functions from the run-time library and class libraries. This is done by the *linker*. Here we discuss the most usual linker operations; a full description of the linker can be found in chapter 12 of the Parallel C *User Guide*[6].

Rather than calling the linker directly, it is usually more convenient to use one of the batch files provided for the purpose.

To link T4 code produced by the `t4cc` compiler use the command:

```
t4cclink object-file
```

For example,

```
t4cclink hello
```

To link T8 code produced by `t8cc` use the command:

```
t8cclink object-file
```

You must use the link command appropriate to the target processor (T4 or T8).

Both batch files automatically append `.bin` to the object file name and produce an executable file with the same file name as the object file and extension `.b4`.

3.2.1 Linking More than One Object File

This section deals with linking more than one object file at a time. If you only want to link single object files for now, you can skip to section 3.3 which describes how to run executable files produced by the linker.

The `t4cclink` and `t8cclink` batch files can be used to link up to nine object files. As before, the extensions of all the object files are assumed to be `.bin`. The executable file generated will have the file name of the first object file specified, with the extension `.b4`.

For example, if there are two C++ source files, `main.cpp` and `fns.cpp`, the following commands will compile them and link them together, producing an executable file for the T4 called `main.b4`.

```
C>t4cc main

C>t4cc fns

C>t4cclink main fns
```

Compiling and linking the example files above for the T8 would be done as follows:

```
C>t8cc main

C>t8cc fns

C>t8cclink main fns
```

3.2.2 Indirect Files

It is quite common for programs to consist of many different object files. The `t4cclink` and `t8cclink` batch files cannot handle more than nine, but even with fewer files than this, you may find the command line awkward to type.

The linker provides a way of getting round this problem, called an *indirect file*. An indirect file is a text file containing a list of object file names, all of which are to be included in the executable file. It is specified in the linker command by its file name preceded by an '`@`'. For example:

```
C>t8cclink @objfiles
```

This will cause the linker to find the file `objfiles.dat`, and link together all the object files specified in it. As usual, the generated file will be given the name of the first object file with the extension `.b4`.

Indirect files are assumed to have the extension `.dat`. They contain a list of MS-DOS file names, with one file name on each line. Full path names, including directory specifications, are allowed. Indirect files may also include the names of other indirect files, by preceding with an '`@`'; nesting indirect files in this way may be done to five levels.

The example indirect file **objfiles.dat** above might contain the following text:

```
main
fns
\userlib\general\io
@grafpack
```

When used in the example given above, this will link the object files **main.bin** and **fns.bin** from the current directory and **io.bin** from the directory **\userlib\general**, together with all the object files specified in the indirect file **grafpack.dat**. The executable file generated will be **main.b4**.

3.2.3 Calling the Linker Directly

Occasionally, instead of using the batch files, you may need to call the linker directly, or write your own batch files to do so. Fuller information about the linker may be found in chapter 12 of the Parallel C *User Guide*[6]. Details of the internal format of object files are provided in the Inmos *Stand-Alone Compiler Implementation Manual*[10].

The linker is invoked by the command `linkt`. The general form of a link command is

`linkt` *object-files* **,** *executable-file*

object-files is a list of object file names separated by spaces. These are the object files which are to be linked together. All of them must have been compiled for the same processor type (T4 or T8). If an object file is specified without an extension, the extension is assumed to be `.bin`.

The order in which the object files are specified is significant. Details of this may be found in sections 3.4 and 4.2.3.2.

The *executable-file* is the name of the file to which the linker writes the executable output code. If no extension is specified, the linker supplies the extension .b4. The executable file and its preceding comma may be omitted; in this case, the executable file is given the same file name as the first object file in the command line, with the extension .b4. If the first file mentioned on the command line is an indirect file, the executable file is given a name taken from the name of the first object file listed in the indirect file.

To link C++ programs, you must include in the list of object files the run-time library, the C++ class library and a special object file called a "harness". The directory `\tc2v2` contains T4 and T8 versions of all these components, as follows:

	T4 version	T8 version
run-time library	`crtlt4.bin`	`crtlt8.bin`
C++ class library	`libct4.bin`	`libct8.bin`
harness	`t4harn.bin`	`t8harn.bin`

The linker will not allow you to mix T4 and T8 object files.

The example below shows the command necessary to link all the files listed in the indirect file `subs.dat` into a single executable file for the T4, called `prog.b4`.

```
C>linkt @subs \tc2v2\libct4 \tc2v2\crtlt4 \tc2v2\t4harn,prog
```

For the T8, the command would be the following.

```
C>linkt @subs \tc2v2\libct8 \tc2v2\crtlt8 \tc2v2\t8harn,prog
```

3.2.4 Libraries

It is often convenient to be able to treat a group of object files as a single unit. For example, the run-time library consists of many separate object files, but is supplied as a single file containing all of them.

The class libraries which are supplied with Parallel C++, such as the Stream Library and the Complex Mathematics Library, are also libraries of this sort.

The linker provides the option of linking together a group of object files to produce a *library* file instead of an executable file. The library contains all of the code and entry points defined by the input object files, which can be changed or deleted without affecting the library. To change a library it must be relinked from its component parts.

Library files have several advantages over using indirect files.

- The linker selects from the library file only those modules which are actually referenced elsewhere in the program; the others are not included in the executable file.
- Copying a single file to another place is simpler than copying many component object files and making sure that the corresponding indirect file is kept up to date with changes in directory and file names.
- Opening just one library file is faster than opening an indirect file and several object files.

However, using an indirect file may be faster while a library is being developed because there is no need to relink the library whenever a component module is changed.

A linker command of the form shown below is used to produce a library from a number of component object files.

`linkt` *object-files*, *library-file*`/l`

The option letter after the '/' is a lower case 'L'.

The form of the input *object-files* is the same as for normal operation of the linker: a list of filenames separated by spaces. Indirect files are indicated by an '@' sign as before.

The *library-file* must be a single MS-DOS file name. If no extension is specified, the linker will give it the extension `.lib`. Note that this is different from the default extension which the linker uses for libraries when they are specified as input files, which is `.bin`.

The example below shows a graphics library being built from a core graphics module and two device driver modules. The library is then linked in the ordinary way with a user program. Indirect files are used to simplify the required linker commands.

```
C>type graflib.dat
core
tek
hp

C>linkt @graflib,graflib.bin/l

C>type myprog.dat
myprog
graflib
\tc2v2\libct8
\tc2v2\crtlt8
\tc2v2\t8harn

C>linkt @myprog
```

3.3 Running

Executable programs are loaded into the transputer board and run using the `afserver` program, which runs on the IBM PC.

The `afserver` is an ordinary MS-DOS program, and after loading the C++ program into the transputer board, it remains active throughout the program's run. Instructions are sent from the run-time library to the `afserver` whenever it needs to perform MS-DOS functions such as reading information from the disks, displaying output on the screen and so on. The results of these operations are sent by the `afserver` back to the transputer board.

The command to load and run a program is:

`afserver -:b` *filename*

The *filename* must be the name of an executable file produced by the linker. The file name extension must be specified. An example of a command to load and run a simple program would be:

```
C>afserver -:b hello.b4
```

Note that this will only work if your program uses a fairly small amount of stack memory. See section 3.4 for information about running programs with larger stack requirements.

Appendix D.3 of the Parallel C *User Guide*[6] includes more information about the `afserver` and its options, and the Inmos *Stand-Alone Compiler Implementation Manual*[10] (section 10) contains a full description. Note that the `-:e` (test error flag) switch described in [10] is not supported for use with Parallel C++ programs. For improved performance, the C++ compiler relies on being able to generate code which might incidentally cause the error flag to be set. Therefore, the transputer error flag may be set as part of the normal execution of a C++ program.

The running of programs can be simplified by putting the appropriate `afserver` command into an MS-DOS batch file. Typing the name of the batch file is then sufficient to run the program. For example:

```
C>type myprog.bat
afserver -:b \mydir\myprog.b4

C>myprog
```

The command `myprog` will then call `afserver` to load the executable file `\mydir\myprog.b4` into the transputer board and start it. Note that if a program compiled and linked for the T4 is loaded into a T8 (or vice versa) the effects will be unpredictable.

3.3.1 Using C++ Programs as MS-DOS Commands

Because of the limitations on what can be done with MS-DOS batch files it is useful to have a way of running a transputer C++ program as if it were an MS-DOS `.exe` file.

You can turn any `.b4` file into an MS-DOS command by making a copy of the file `\tc2v2\linkt.exe` in the same directory as the `.b4` file, giving it the same root filename as the `.b4` file but keeping the `.exe` extension. For example, if the current directory contains the executable file `calc.b4`, it can be run as a command by typing:

```
C>copy \tc2v2\linkt.exe calc.exe

C>calc
```

This new `calc` command can be used from any directory, provided the directory containing `calc.exe` and `calc.b4` is on the MS-DOS search path.

(`linkt.exe` works by taking the command verb from its command line, adding `.b4`, and then calling `afserver` to load that file from the same directory `linkt.exe` itself was loaded from).

When a `.b4` file is invoked via a "driver" program in this way, the `-:o 1` option (see section 3.3) is added automatically and the program is given a large amount of stack space. If you want to run a program as an MS-DOS command, but with its stack in fast on-chip RAM, you should invoke the program as usual but add `-:o 0` to the command line (hyphen, colon, letter 'o', then a space followed by the digit zero). For example:

```
C>ex -:o 0
```

3.3.2 Command-Line Arguments

The `afserver` passes its command line on to the user program it invokes, for use as program arguments. For example:

```
C>afserver -:b myprog.b4 fred
```

Here, the character string "`fred`" is passed on to `myprog.b4`.

Note that the "`-:b myprog.b4`" part of the command is not passed through as an argument to `myprog.b4`. In general, `afserver` option switches (`-:b`, `-:o`) and their arguments are not passed on to the user program. Any MS-DOS file redirections (see section 3.3.3 below) are also stripped out.

The text of the command line is also passed on to the user program if the `afserver` is invoked using the driver program described in section 3.3.1. For example:

```
C>myprog xyz abc
```

Here, the program argument string "`xyz abc`" is passed on to `myprog.b4`.

The program argument string is broken up into a sequence of tokens before being passed to the C++ `main` program function. Tokens are separated by blank or horizontal tab characters, so in the first example there was one token: "`fred`", and in the second example there were two: "`xyz`" and "`abc`".

When the C++ `main` program function is called, it is passed the following arguments:

```
main(int argc, char *argv[])
```

`argv[0]` is the program name, currently always a pointer to a null string (i.e., a pointer to a '\0' character).

If the value of `argc` is greater than one then `argv[1]`... `argv[argc-1]` are pointers to token strings each of which is terminated by '\0'.

`argv[argc]` is a null pointer.

`argc` is the number of tokens, including the program name. It is always greater than zero.

3.3.3 I/O Redirection and Piping

Normally the C++ standard input stream `cin` is associated with the keyboard. Standard input can be taken from a file by using the MS-DOS redirection symbol '<' in the normal way. For example, to use the file `chap1.txt` as the standard input stream for a word counting program `wc.b4` you could use the command:

```
C>afserver -:b wc.b4 <chap1.txt
```

This also works if `wc.b4` is invoked by a driver program, `wc.exe`:

```
C>wc <chap1.txt
```

Similarly, the standard output stream `cout` is normally associated with the screen. Standard output is redirected using the '>' symbol. A program called `cat.b4` which concatenated the contents of all the input filenames given as its program arguments and wrote the result to the standard output stream could be used to concatenate the files `a.txt`, `b.txt` and `c.txt`, writing the result to another file `all3.txt` as follows:

```
C>afserver -:b cat.b4 a.txt b.txt c.txt >all3.txt
```

Note that neither ">*filename*" nor "<*filename*" is considered to be part of the program arguments; these special forms do not appear in the `argv` array passed to a C++ main program.

Standard output may also be *piped* into an MS-DOS *filter* program by writing the name of the filter after a vertical bar '|', as shown below.

```
C>afserver -:b cat.b4 a.txt b.txt | more
```

The *DOS Reference Manual* describes in detail what can be done with filters. (The `more` program simply displays its input on the screen, a page at a time).

3.4 Memory Use

The memory used by a C++ program is divided into four storage areas.

- *Code storage* is used to hold the executable instructions of the program itself, together with some constant data and control information.

- *Static storage* is used to hold static and external variables, including variables declared at the global level.

- *Stack storage*(sometimes referred to as *workspace*) is used for auto variables. The stack is also used for function calls and passing parameters.

  In addition, library functions use varying amounts of stack space as working storage. The stack requirements of the mathematical functions are given in the Inmos *TDS Compiler Implementation Manual*[11] (Section 10, Parameters and workspace requirements) and are generally about 40 to 100 words. The stack requirements of the floating-point arithmetic support library for the T4 are generally about 10 to 40 words. About 70 words of stack storage are permanently reserved for use by the run-time library.

- *Heap storage* is used to hold all variables created by `new`, etc. It is also used internally by the run-time library for I/O buffers, etc.

These four areas of storage are mapped onto two areas of physical memory:

- *On-chip* memory. The T4 has 2KB of fast on-chip memory, and the T8 has 4KB.

- *External* memory. The Inmos B004 board has either 1MB or 2MB of external memory.

Using the linker only, two methods of mapping the storage areas onto physical memory are available: the default method, and the alternative method. You can select the method you wish to use by calling the **afserver** in different ways, which are discussed below.

The configurers required for developing parallel programs give the user more advanced methods for controlling the use of memory.

3.4.1 Default Memory Mapping

Default memory mapping is used if the **afserver** program is called as described in section 3.3 above. With this arrangement, the T4's on-chip memory, and the first 2KB of the T8's on-chip memory, are used for stack storage. Since on-chip memory is faster than external memory, programs can run much faster with default memory mapping. Obviously, you must be certain that the program's stack storage will fit in the available 2KB.

If you are using a T8, default memory mapping provides an opportunity for further speed improvements, since the remaining 2KB of the T8's on-chip memory is available for code storage. To take advantage of this, you should place small, speed-critical functions at the beginning of the link-list.

WARNING: A program which exceeds the amount of available stack space will fail in unpredictable ways: it may hang, or it may simply give wrong answers.

3.4.2 Alternative Memory Mapping

Unless you are sure your program's stack data will fit into the 2KB of available on-chip memory, you should use the alternative method of memory mapping. This is done by calling the **afserver** like this:

```
C>afserver -:b myprog.b4 -:o 1
```

With the alternative method, the stack is placed in external memory, and so is limited only by the amount of external memory available. On the T4, on-chip RAM is not used at all. On the T8, although the upper 2K of on-chip RAM is used for code as before, the rest of it is unused.

The program will execute more slowly with this method, because external memory is slower than on-chip memory.

Note that the `afserver` switch is typed as hyphen, colon, option letter 'o', then a space, then the digit one.

3.4.3 Limit on Program Memory

The current version of the linker generates executable files which will only run correctly on boards having 1MB or 2MB of memory. To get round this restriction, the Parallel C++ environment includes the `mempatch` program which may be used to change executable files to run on boards which have different amounts of memory. See chapter 13 of the Parallel C *User Guide*[6] for a discussion of `mempatch`.

3.5 Parallel Programming

The facilities for parallel programming provided with Parallel C, and described in chapters 4 and 5 of the Parallel C *User Guide*[6], are all also applicable to Parallel C++. This includes the configurers, and the run-time library support for threads, channel and link operations, semaphores, timers, `alt` functions and functions for accessing MS-DOS facilities. In addition, Parallel C++ programs can make use of the file-service multiplexer, as described in chapter 6 of the Parallel C *User Guide*, and Parallel C++ can be used to build processor farm applications, as described in chapter 8.

```
// driver.cpp:  driver for uppercasing example

#include <chan.h>
#include <iostream.h>

void main (int argc, char *argv[], char *envp[],
           CHAN *in_ports[], int ins, CHAN *out_ports[], int
outs)
{
   int c;
   for (;;) {
      c = cin.get();
      chan_out_word(c, out_ports[2]);
      if (c == EOF) break;
      chan_in_word(&c, in_ports[2]);
      cout.put(char(c));
   }
}
```

Figure 3.1: Driver for uppercasing example

Note. The current version of the Complex Mathematics Library (see chapter 5) cannot be linked with stand-alone tasks.

3.5.1 Building Parallel Programs

Let us consider a C++ version of the two-task uppercasing example discussed in sections 5.2 and 5.3 of the Parallel C *User Guide*.

First we have the driver task, shown in figure 3.1. This can be compiled in the usual way, and must then be linked with the task harness, rather than the standard harness (see section 5.1.3.1 of the Parallel C *User Guide*). For the T4, this would be done as follows:

```
C>t4cc driver

C>t4cctask driver
```

```
// upc.cpp: processing task for uppercasing example

#include <chan.h>
#include <ctype.h>
#define EOF (-1)

void main (int argc, char *argv[], char *envp[],
          CHAN *in_ports[], int ins, CHAN *out_ports[], int
outs)
{
   int c;
   for (;;) {
      chan_in_word(&c, in_ports[0]);
      if (c == EOF) break;
      chan_out_word(toupper(c), out_ports[0]);
   }
}
```

Figure 3.2: Processing task for uppercasing example

For the T8, the procedure would be:

```
C>t8cc driver

C>t8cctask driver
```

Next, let us look at the processor task, which is shown in figure 3.2. As we can see, the task contains no calls on the stream library. This is correct, as the processing task will be *stand alone*, that is, without `afserver` support, and consequently cannot perform standard C or C++ I/O. Such tasks are linked with a special stand-alone version of the C library. A T4 version of the processing task can be built with these commands:

```
C>t4cc upc

C>t4ccstask upc
```

The batch file `t4ccstask` links the program with the C stand-alone run-time library and the task harness. As usual, there is a T8 version: `t8ccstask`.

Finally, the two-task application can be built, using the configurer and a configuration file. This is done in exactly the same way as the corresponding C example.

3.5.2 Synchronising Access to the Libraries

In sections 5.6.1 and 10.11, the Parallel C *User Guide* discusses an important problem which arises when a program has more than one thread active. This is the possibility that more than one thread may try to access the same part of the run-time library at the same time.

To avoid this happening, we have to make sure that the threads interlock their access to the run-time library, using the semaphore `par_sema`. If this is not done, a program is likely to fail in unpredictable ways. You should take particular care with the following two Parallel C++ facilities.

- The functions of the stream library often perform I/O, and should therefore be interlocked. For example:

  ```
  #include <par.h>
  #include <iostream.h>
  :
     sema_wait(par_sema);
     // construct a file stream and open file
     infile fstream("input.dat", ios::in);
     sema_signal(par_sema);
  ```

- The `new` or `delete` operators perform operations on the heap, and must be interlocked. At the user level, the problem can be avoided by overloading the `new` and `delete` operators to use the `par_malloc` and `par_free` functions. Programmers should be aware, however, that many members of many classes, including classes in the Complex and Stream libraries, use `new` and `delete`.

In general, concurrent threads in Parallel C++ must be treated with great care.

Part III

Reference

Chapter 4

C++ Compiler Reference

This chapter contains technical information about the way the C++ language is implemented on the transputer. Note that the information in this chapter applies only to the current version of the compiler; it is not guaranteed that future versions of the compiler will behave in the same way.

It should be useful to read this chapter in conjunction with chapter 9 of the Parallel C *User Guide*[6]. Information contained in that chapter will not be repeated here.

4.1 Running the Compiler

The compiler is run by one of the commands `t8cc` or `t4cc`.

`t8cc` generates object code for the T800 floating-point transputer.

`t4cc` generates object code for the T414 32-bit transputer.

The command line used to invoke the compiler must specify a single source file name. Wild cards are not allowed. If no extension is specified, `.cpp` is assumed.

Option switches may optionally be given on the command line. Option switches are introduced by the '/' character; the available switches are discussed in section 4.2 below.

If the source file is successfully compiled, a zero exit status code is returned to MS-DOS. If errors are detected, the compiler returns an exit status code of 1. This feature can be used in MS-DOS batch files to check whether a compilation was successful.

The compiler creates a number of temporary files as it works. Normally, these are placed in the current directory; however, the environmental variable `TMP` may be used to make the compiler put them in another directory. For example, to make the compiler place the temporary files in the root directory on disk `D:`, the following MS-DOS command could be used.

```
C>set TMP=D:\
```

The names of the temporary files either start with the string **$3L$**, or they are of the form `ctemp.`*n*, where *n* is a small integer. Usually, the compiler will delete them at the end of the run, but occasionally this may not be done; in this case, it is safe to delete them yourself.

4.2 Compiler Switches

This section describes the switches available to control the behaviour of the compiler. Switches are introduced by a '/' character and may be typed in any order, before or after the source file specification. Except as noted below, switches and their argument strings are not case-sensitive; that is, lower-case letters have the same significance as the corresponding upper-case letters. This means, for example, that the following two switches would be treated the same:

```
/FBhello.bin
/fbHELLO.BIN
```

The format of the various switches is described using the following notations:

fn — An MS-DOS filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 4.2.1 below.

dir — An MS-DOS filename, which will be assumed to refer to a directory.

mac — Any sequence of characters which is acceptable to the compiler as a macro name.

str — Any sequence of characters which is acceptable to the compiler as the value of a macro.

n — A decimal integer.

An example of a command to invoke the compiler with switches:

```
C>t8cc hello /dLEVEL=3 /fbkeep /i
```

This will invoke the T8 compiler to compile `hello.cpp`, and place the binary output in `keep.bin`. Before the compilation, a macro `LEVEL` will be defined with the value 3. Details of the identities and versions of the compiler components will be printed.

4.2.1 Controlling the Object File

4.2.1.1 Switches /FB and /FO

The `/FB` or `/FO` switch is used to specify the name of the object file output by the compiler. The two switches have the same effect.

The switch must be followed by a *fn*, but the complete MS-DOS path name may not be necessary. The compiler supplies defaults, as follows:

- If no extension is given, the compiler supplies the default extension `.bin`.
- If no filename is given, the filename of the source file is used.

- If the drive specification or directory specification are omitted, then the current drive and/or directory are used.

- If a drive specification is given alone, then the output file is created in the current directory of the specified drive, regardless of the source file's directory.

The following examples may clarify this. The 'Supplied' string below is assumed to be the argument of a `/FB` switch. The current drive and directory are `c:\michael`, and the current directory on `a:` is `\output`.

Specified source file	Supplied	Output file
dogs	*nothing*	`c:\michael\dogs.bin`
dogs	`cats`	`c:\michael\cats.bin`
dogs	`cats.out`	`c:\michael\cats.out`
dogs	`\stuff\`	`c:\stuff\dogs.bin`
dogs	`a:\first\`	`a:\first\dogs.bin`
dogs	`a:`	`a:\output\dogs.bin`
dogs	`a:cats`	`a:\output\cats.bin`

Notice that in examples like the fourth above, it is the fact that the supplied string ends with a '\' which indicates that this is a directory specification. If it is omitted, output would be sent (in this case) to `c:\stuff.bin`, even if a directory `c:\stuff` exists.

If no `/FB` or `/FO` switches are specified, the behaviour of the compiler is the same as if a `/FB` switch were used, with no argument. In order to stop the compiler generating an object file of any kind, the `/C` switch must be used (see section 4.2.2).

4.2.2 Controlling Object Code

4.2.2.1 Switch /Gd

By default, the compiler follows the ANSI standard in using single-precision floating-point arithmetic when both operands of an arithmetic operator are of type `float`.

The /Gd switch is provided so that the compiler can be made to follow the earlier K&R rule, if necessary. *The C Programming Language*[5] states that "all floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double`...". This means that an expression like `a+b`, where `a` and `b` are `float`, is evaluated by first converting `a` and `b` to `double` and then performing the addition using double-precision floating-point arithmetic.

The ANSI practice results in faster program execution, but because floating-point arithmetic works with approximations the numerical result of the operation may be less accurate than that obtained before. Users who are affected by this may prefer to use the /Gd switch.

Note that even without /Gd, floating-point constants are still `double`, and so an expression like `2.0*a` will still be evaluated in double precision (with `a` being converted to `double`). You can avoid this happening by assigning the value `2.0` to a `float` temporary variable beforehand (`two` say) and then writing the expression as `two*a`.

4.2.2.2 Switch /C

If this option switch is used, the compiler checks the source file for errors, but does not generate an object file.

4.2.3 Controlling Code Patch Sizes

Certain constant values in a program cannot be worked out by the compiler, but must be filled in (or *patched*) by the linker. The compiler leaves gaps for these values, and fills the gaps with a special code. In some circumstances, however, the linker may decide on a patch value which is too large to fit in the gap provided by the compiler. When this happens, the linker gives the following error message:

```
FATAL ERROR(22): patch over valid code in module module
```

The /P switch controls the sizes of the gaps left by the compiler, so that this situation can be avoided. There are two varieties.

4.2.3.1 Switch /PCn

This switch changes the size of the gap the compiler leaves for a function call. The size of the gap limits the distance from the call to the called function. Four bits of the ***displacement*** are stored in every byte of gap, so the maximum displacement is $2^{4n} - 1$ bytes. n should be in the range 2 to 8. If the /PC switch is not used, the compiler assumes a value of 6 for n, giving a maximum displacement of 16MB. Similar negative displacements are also allowed. Smaller values of n reduce the code size for external calls (resulting in faster execution) but restrict the total size of the final program image. For example, $n = 5$ allows displacements up to 1MB; $n = 4$ allows up to 64KB. Normally the default value of n should be adequate.

The compiler does not accept a /PC1 switch, as in this case not only would the displacement be restricted to 15 bytes, but in addition backward calls would not be possible.

4.2.3.2 Switch /PM*n*

A linked program contains a *module table*, which has an entry for every module in the program, including both the modules written by the user and those extracted from libraries. Each module's entry contains the address of the module's static data area. The first thing which a subprogram does is to access this address, and to do this, it must load the *module number*. These module numbers are assigned by the linker, so the compiler cannot predict how large a module's number will be. Once again, it leaves a gap, and the /PM switch allows the user to specify how large this gap is. Four bits of the module number are stored in every byte of gap, so the maximum module number is $2^{4n} - 1$ bytes. n should be in the range 2 to 8. If the /PM switch is not used, the compiler assumes a value of 2 for n, giving a maximum module number of 255. Larger /PM numbers increase the maximum number of modules which can be linked into one program, but make the program slightly larger and slower.

If the linker reports `patch over valid code`, as described above, the likely cause is that the linked program contains more than 255 modules, including library modules. The programmer can cope with this situation as follows:

- Use /PM to increase the maximum allowable module number. For example, /PM3 will allow 4096 modules.
- Modules are assigned numbers in order, depending on their position in the linker's command line. It is essential that modules from the run-time library and C++ class libraries should have module numbers which are less than 255; they have already been compiled with /PM2, and this cannot be changed. So the linker command line should have these libraries and the harness first; then any user-written modules and libraries, compiled with a larger /PM. For example:

```
C>linkt \tc2v2\libct8 \tc2v2\crtlt8 \tc2v2\t8harn main
@mysubs,main.b4
```

4.2.4 Controlling Debugging

The following switches control the output of information required by the decode program and by Tbug, 3L's interactive symbolic debugger for the transputer.

4.2.4.1 Switch /Zd

This switch causes the compiler to include line-number tables in the generated object file. These tables are used by decode and by Tbug to work out which piece of object code corresponds to each line of the source program. If this switch is not used, this information will not be available, and Tbug will not be able to display the source version of the program.

4.2.4.2 Switch /Zi

This switch causes the compiler to include variable tables in the generated object file. These contain information about the names, locations and types of the program's identifiers. If this switch is not used, Tbug will not be able to display the variables by name and in the correct format.

The /Zi switch will also cause the compiler to output the line-number tables. This means that if you use /Zi, you do not need to use /Zd as well.

4.2.4.3 Switch /Zo

This switch causes the compiler to generate diagnostic information in an older format which is not required for use with Tbug. This facility is retained in order to maintain compatibility with the 3L system programming environment, and is unlikely to be needed by end-users.

4.2.5 Controlling #include Processing

This section should be read in conjunction with section 4.4, where include file processing is discussed more fully.

4.2.5.1 Switch /I*dir*

This switch adds *dir* to the include list, that is, the list of "standard places" where the compiler looks for files specified in `#include` lines. The *dir* string is assumed to be a directory, whether or not it terminates with a '\'.

4.2.6 Macro Definitions

This section should be read in conjunction with section 4.3, where predefined macros are discussed.

4.2.6.1 Switch /D*mac* and /D*mac=str*

The first form of the /D switch can be used to define a macro with the value '1'. The second form enables the user to define a macro with the value '*str*'. These definitions are done before the compilation of the program. For example:

```
C>t8cc/dDEBUG/Dhelp=3/dJOE=Jim cats
```

This is equivalent to coding the following lines at the top of the program `cats.cpp`:

```
#define DEBUG 1
#define help  3
#define JOE   Jim
```

Notice that the macro names and their values are case sensitive. If there are any syntax errors in the definitions, these are reported on the display and included on the listing (if any) in the usual way.

4.2.6.2 Switch /U*mac*

This switch undefines a predefined macro—see section 4.3 for a discussion of these. This means, for example, that the following switch:

```
C>t8cc/U_transputer cats
```

is equivalent to coding the following line at the top of **cats.cpp**:

```
#undef _transputer
```

Once again, the name of the macro is case sensitive.

4.2.7 Information from the Compiler

4.2.7.1 Switch /I

This switch makes the compiler display information about itself, including the identities and version numbers of its components. Please quote this information in any correspondence about the compiler.

4.2.7.2 Switch /W

The /W switch controls the output of warning messages from the compiler. Without this switch, the compiler warns about constructs which are likely to be mistakes, non-portable or inefficient. If /W is specified, the compiler will only issue warnings about constructs which are almost certainly errors.

4.3 Predefined Macros

The following macros are defined with the value '1' for every compilation:

```
_transputer
_3L
```

```
CII
__cplusplus
```

The following two macros are defined to indicate which processor the current compilation is for:

`_IMST4`	for compilations by `t4cc`
`_IMST8`	for compilations by `t8cc`

Any of these predefinitions may be cancelled by the `/U`*mac* switch. See section 4.2.6 for details.

4.4 Handling of #include Files

When the compiler encounters an **#include** line, it searches for the specified file in a sequence of directories known as the *include list*. This consists of the following, which are searched in this order:

1. The current directory—except in the case of lines of this format:

   #include <*filename*>

2. Directories which have been specifically added to the include list at compilation time by means of the `/I` switch—see section 4.2.5.

3. The directory `\tc2v2\cc`.

Chapter 5

The C++ Complex Mathematics Library

5.1 Introduction to the `complex` class

This chapter describes the facilities of the Parallel C++ Complex Mathematics Library.

In order to make use of these facilities, the program must include the following line:

```
#include <complex.h>
```

The file `complex.h` includes the header file `math.h`.

The Complex Mathematics Library is not automatically searched by the linker. If your program makes use of this library, you must include in your link-list the file `\tc2v2\complxt8` (for T8 programs) or `\tc2v2\complxt4` (for T4 programs). For example, to link a program `calc` with the T8 version of the Complex Mathematics Library, you should give this command:

```
C>t8cclink calc \tc2v2\complxt8
```

Note. The current version of the Complex Mathematics Library cannot be linked with stand-alone tasks. See section 3.5.

5.1.1 The `complex` Class

The Complex Mathematics library implements the data type of complex numbers as a class, `complex`. It overloads the standard input, output, arithmetic, assignment, and comparison operators, discussed in section 5.3. Routines for converting between Cartesian and polar coordinate systems are discussed in section 5.4. The `complex` class also overloads the standard exponential, logarithm, power, and square root functions, discussed in section 5.5, and the trigonometric functions of sine, cosine, hyperbolic sine, and hyperbolic cosine, discussed in section 5.6.

Error handling for the `complex` class functions is described in the next section.

Constructor

`complex`	constructor function for `complex` objects

```
complex(double re, double im);
```

The argument `re` specifies the real part of the complex value, and `im` specifies the imaginary part. If the `im` argument is omitted, the imaginary part is given the value 0.0.

5.2 Error Handling

5.2.1 Default Error Handling

Certain functions in the Complex Mathematics Library may result in a value which is undefined for the given arguments, or which is not representable. These errors are classified into the following types.

`SING` Argument singularity

`OVERFLOW` Overflow range error

`UNDERFLOW` Underflow range error

The following table describes the way in which these error types are handled by default.

	Error Type		
	`SING`	`OVERFLOW`	`UNDERFLOW`
`errno`	`EDOM`	`ERANGE`	`ERANGE`
function `exp`:			
real too large/small	-	$(\pm H, \pm H)$	$(0, 0)$
imag too large	-	$(0, 0)$	-
function `log`:			
argument = $(0, 0)$	M, $(H, 0)$	-	-
function `sinh`:			
real too large	-	$(\pm H, \pm H)$	-
imag too large	-	$(0, 0)$	-
function `cosh`:			
real too large	-	$(\pm H, \pm H)$	-
imag too large	-	$(0, 0)$	-

The notations in the table for the error actions have the following meanings.

M Message is printed (`EDOM` error).

$(H, 0)$ (`HUGE_VAL`, 0) is returned.

$(\pm H, \pm H)$ ($\pm$`HUGE_VAL`, $\pm$`HUGE_VAL`) is returned.

$(0, 0)$ (0, 0) is returned.

The macro `HUGE_VAL` is defined in the header file `math.h`, which is included in the program by `complex.h`.

5.2.2 Trapping Errors

The default error-handling described above is performed by the library function `complex_error`. If you wish to trap these error conditions and handle them in the program, you need to write a new `complex_error` function yourself.

The header `complex.h` includes a definition of the class `c_exception`, as follows.

```
class c_exception
{
   int      type;
   char     *name;
   complex  arg1;
   complex  arg2;
   complex  retval;

public:
   c_exception( char *n, const complex& a1,
                const complex& a2 = complex_zero );
   friend int complex_error( c_exception& );
   friend complex exp ( complex );
   friend complex sinh ( complex );
   friend complex cosh ( complex );
   friend complex log ( complex );
};
```

The data elements of the `c_exception` are used as follows.

`type`	The type of the error: one of the three constants `SING`, `OVERFLOW` and `UNDERFLOW`, which are defined in `complex.h` and described in the previous section.
`name`	Points to a string containing the name of the function which has encountered the error.
`arg1, arg2`	The arguments with which the function was invoked.
`retval`	The value of the function which will be returned to the user.

Constructor

c_exception	c_exception constructor

```
c_exception(char *n, const complex& a1,
        const complex& a2 = complex_zero);
```

The argument n points to a string containing the name of the function. The arguments a1 and a2 are references to arguments with which the function was invoked; the default value for a2 is (0,0).

Function

complex_error	error handling function

```
friend int complex_error(c_exception& c);
```

As we remarked above, the complex library contains a default version of this function. A new version should be written by users who wish to handle their own complex errors.

The argument c is a reference to a c_exception object, whose data elements contain the details of the error to be handled. The new complex_error function should place in the element retval the value to be returned to the main program by the function which has detected an error.

If the value returned by the complex_error function is non-zero, no error message will be printed.

5.3 Operators

The basic arithmetic operators, comparison operators, and assignment operators are overloaded for complex numbers. The operators have their conventional precedences.

Arithmetic Operators

The four basic binary arithmetic operators are overloaded for complex values, as is the unary '-' operator. For example, the following example is valid.

```
complex z, a, b, c, d;
z = (a + b*c) / -d;
```

operator: +	complex addition

```
friend complex operator+(complex x, complex y);
```

Returns a complex value which is the arithmetic sum of **x** and **y**.

operator: -	complex negation

```
friend complex operator-(complex x);
```

Returns a complex value which is the arithmetic negation of **x**.

operator: -	complex subtraction

```
friend complex operator-(complex x, complex y);
```

Returns a complex value which is the result of subtracting **y** from **x**.

operator: *	complex product

```
friend complex operator*(complex x, complex y);
```

Returns a complex value which is the arithmetic product of **x** and **y**.

operator: /	complex division

```
friend complex operator/(complex x, complex y);
```

Returns a complex value which is the result of dividing x by y.

Comparison Operators

The operators for testing for equality and inequality are overloaded for complex values. For example, the following is allowed.

```
complex a, b;
if (a != b && a == complex (0,0)) {
   cout << "\nokay";
}
```

operator: ==	complex equality

```
friend int operator==(complex a, complex y);
```

Returns non-zero if x is equal to y; returns 0 otherwise.

operator: !=	complex inequality

```
friend int operator!=(complex x, complex y);
```

Returns non-zero if x is not equal to y; returns 0 otherwise.

Assignment Operators

The operators +=, -=, *= and /= are overloaded for assigning complex values to complex objects. For example, the following is valid:

```
complex x, y;
x += y;
x -= y;
```

```
x *= y;
x /= y;
```

It is important to note the complex assignment operators do not produce a value that can be used in an expression, unlike the "built-in" C assignment operators. This means, for example, that the following construction is syntactically invalid.

```
complex x, y, z;
x = ( y += z );
```

operator: +=	add and assign

```
void operator+=(complex y);
```

The value y is added to the complex object on the left of the operator.

operator: -=	subtract and assign

```
void operator-=(complex y);
```

The value y is subtracted from the object on the left of the operator.

operator: *=	multiply and assign

```
void operator*=(complex y);
```

The object on the left of the operator is multiplied by the value y.

operator: `/=`	divide and assign

```
void operator/=(complex y);
```

The object on the left of the operator is divided by the value y.

5.4 Cartesian/Polar Functions

This section discusses functions for conversions between the Cartesian and polar coordinate systems.

abs	complex absolute value

```
friend double abs(complex x);
```

abs returns the absolute value (or magnitude) of x.

norm	square of the magnitude

```
friend double norm(complex x);
```

norm returns the square of the magnitude of **x**. It is faster than **abs** but more likely to cause an overflow error. It is intended for comparison of magnitudes.

arg	angle

```
friend double arg(complex x);
```

arg returns the angle of **x** measured in radians, in the range $-\pi$ to $+\pi$.

conj	complex conjugate

```
friend complex conj(complex x);
```

`conj` returns the complex conjugate of `x`. For example, if `x` is `(real, imag)`, `conj(x)` is `(real, -imag)`.

polar	polar coordinates

```
friend complex polar(double m, double a = 0);
```

`polar` returns a complex value given a pair of polar coordinates, magnitude `m`, and angle `a`, measured in radians in the range $-\pi$ to $+\pi$. If the argument `a` is not supplied, a value of 0 is assumed.

real	real part

```
friend double real(complex x);
```

`real` returns the real part of the complex argument `x`.

imag	imaginary part

```
friend double imag(complex x);
```

`imag` returns the imaginary part of the complex argument `x`.

5.5 Mathematical Functions

The **complex** class includes overloadings for complex arguments of the functions **exp**, **log**, **pow** and **sqrt**.

exp	e^x function

```
friend complex exp(complex x);
```

exp returns the complex value $e^{\mathbf{x}}$.

exp returns (0,0) when the real part of **x** is so small, or the imaginary part is so large, as to cause overflow. When the real part is large enough to cause overflow, **exp** returns (HUGE_VAL,HUGE_VAL) if the cosine and sine of the imaginary part of **x** are positive, (HUGE_VAL,-HUGE_VAL) if the cosine is positive and the sine is not, (-HUGE_VAL,HUGE_VAL) if the sine is positive and the cosine is not, and (-HUGE_VAL,-HUGE_VAL) if neither sine nor cosine is positive. In all these cases, **errno** is set to ERANGE. You can change this treatment of exceptional cases by writing your own version of **complex_error**; see section 5.2.

log	$\log_e x$ function

```
friend complex log(complex x);
```

log returns the natural logarithm of **x**.

If **x** is (0,0), **log** returns (-HUGE_VAL,0) and sets errno to EDOM. A message indicating SING error is printed on the standard error output. You can change this procedure by writing your own version of **complex_error**; see section 5.2.

pow	calculates x^y

```
friend complex pow(complex x, complex y);
```

pow returns the complex value of x raised to the power of y.

sqrt	calculates $\sqrt{x}$

```
friend complex sqrt(complex x);
```

sqrt returns the square root of x, contained in the first or fourth quadrants of the complex plane.

5.6 Trigonometric and Hyperbolic Functions

This section describes the overloading of trigonometric and hyperbolic functions for complex values.

Functions

sin	sine function

```
friend complex sin(complex x);
```

sin returns the sine of of its radian argument.

cos	cosine function

```
friend complex cos(complex x);
```

cos returns the cosine of its radian argument.

sinh hyperbolic sine function

```
friend complex sinh(complex x);
```

sinh returns the hyperbolic sine of its argument.

sinh() returns **(0,0)** if the imaginary part of **x** would cause overflow. When the real part is large enough to cause overflow, **sinh()** returns **(HUGE_VAL,HUGE_VAL)** if the cosine and sine of the imaginary part of **x** are non-negative, **(HUGE_VAL,-HUGE_VAL)** if the cosine is non-negative and the sine is less than **0**, **(-HUGE_VAL,HUGE_VAL)** if the sine is non-negative and the cosine is less than **0**, and **(-HUGE_VAL,-HUGE_VAL)** if both sine and cosine are less than **0**. In all these cases, **errno** is set to **ERANGE**. You can change this treatment of exceptional cases by writing your own version of **complex_error**; see section 5.2.

cosh hyperbolic cosine function

```
friend complex cosh(complex x);
```

cosh returns the hyperbolic cosine of its argument.

cosh() returns **(0,0)** if the imaginary part of **x** would cause overflow. When the real part is large enough to cause overflow, **cosh()** returns **(HUGE_VAL,HUGE_VAL)** if the cosine and sine of the imaginary part of **x** are non-negative, **(HUGE_VAL,-HUGE_VAL)** if the cosine is non-negative and the sine is less than **0**, **(-HUGE_VAL,HUGE_VAL)** if the sine is non-negative and the cosine is less than **0**, and **(-HUGE_VAL,-HUGE_VAL)** if both sine and cosine are less than **0**. In all these cases, **errno** is set to **ERANGE**. You can change this treatment of exceptional cases by writing your own version of **complex_error**; see section 5.2.

Chapter 6

The Parallel C++ Stream Library

6.1 Introduction

This chapter describes the Parallel C++ stream package. Although all the facilities of the package are dealt with here, as usual this is not intended as a tutorial description, and the reader is referred to one of the standard texts for a more easily-assimilable discussion.

The package is declared in `iostream.h` and a number of other header files, which are listed in section 6.1.4. It consists primarily of a collection of classes. Although originally intended only to support input/output, the package now supports related activities such as "in-store" formatting.

The stream package implemented in Parallel C++ is a mostly source-compatible extension of the earlier stream I/O package described in *The C++ Programming Language*[1].

Note

In this chapter, the word *character* is used to refer to a value that can be held in either a `char` or `unsigned char`. When functions that return an `int` are said to return a character, they return a positive value. Usually such functions can also return `EOF` as an error indication.

As usual, the word *byte* refers to the piece of memory that can hold a character. Thus, either a `char*` or an `unsigned char*` can point to an array of bytes.

6.1.1 Buffers and Streams

Input/output in C++ involves operations on two kinds of objects: *streams* and *buffers*.

Buffer objects support the following operations.

- *Insertion* (also called *storing* or *putting*) of characters into a *sink*. Sinks include MS-DOS standard streams (for example `stdout`), files or arrays.

- *Extraction* (also called *fetching* or *getting*) of characters from a *source*. Sources include the MS-DOS standard stream `stdin` as well as files or arrays.

- Some buffer classes also support operations such as the closing, opening and positioning of files.

Stream objects support formatted and unformatted conversion of sequences of characters which are stored in or fetched from buffers, as well as the other operations supported by buffer objects.

For the most part, users will not need to perform operations on buffers, but will use the associated stream objects instead.

6.1.2 Classes

6.1.2.1 Base Classes

Most users of the stream package will not need to operate on objects of these classes directly. Those who wish to extend the package with new stream and buffer classes, however, will need to read section 6.7, where the public and protected interfaces of the `streambuf` class are discussed.

`streambuf` This is the base class for buffers. It supports the fundamental operations on buffers, including insertion and extraction. Most members of `streambuf` are inlined for efficiency.

`ios` This is the base class for streams. It contains various variables which define the current state of a stream, such as its error and formatting states. It also contains certain `enum` definitions which are used as formatting manipulators, open modes and so on.

6.1.2.2 Core Stream Classes

Objects of these classes support the basic stream facilities. These are performed in each case by operations on an associated object of class `streambuf`. These classes are also used as base classes for the stream objects described later in this section.

The facilities supported by these streams are described in detail in sections 6.2 and 6.3.

`istream` This class supports formatted and unformatted conversion of sequences of characters fetched from the associated `streambuf`. The `>>` operator is overloaded to perform an extraction.

`ostream` This class supports formatted and unformatted conversion of sequences of characters which are stored in the associated `streambuf`. The `<<` operator is overloaded to perform an insertion.

`iostream` This class derives from `istream` and `ostream`, and is intended for situations when both input and output (extraction and insertion) of sequences of characters is needed. This class is used mostly as a base for the `fstream` class discussed below.

`istream_withassign`, `ostream_withassign`, `iostream_withassign`
These classes derive from the corresponding classes without the `_withassign` suffix. They add assignment operators to these classes, and also implement a constructor which has no operands. The predefined streams `cin`, `cout`, `cerr` and `clog` are objects of these classes; for a discussion of these streams, see section 6.1.3 below.

6.1.2.3 Operations on Files

These classes support input/output on MS-DOS files. For a full description of the facilities supported, see section 6.4.

`filebuf` This class is derived from `streambuf`. Members support opening, closing and seeking. Most users will not need to manipulate objects of this class directly, but will use an associated stream instead.

`ifstream` This class supports input from files, by performing formatted and unformatted conversion of sequences of characters fetched from an associated `filebuf`. It is derived from `istream`, and so supports all the formatting facilities of that class.

ofstream In the same way, this class is derived from **ostream**, and supports output to files by storing characters in an associated **filebuf**.

fstream This class is derived from **iostream**, and is used when you need to perform input and output on the same file.

6.1.2.4 Operations on Arrays

These classes support "in-store" formatting. For a full description, see section 6.5.

strstreambuf
This class, which is derived from **streambuf**, supports insertion and extraction operations on arrays of bytes in memory. As usual, most users will not need to manipulate objects of this class directly.

istrstream This class allows the user to fetch characters from an array of bytes and convert them using the standard stream facilities. It is derived from **istream**.

ostrstream In the same way, this class, which is derived from **ostream**, allows the user to convert data into sequences of characters with are stored in an array.

6.1.2.5 Operations on FILE Structures

These classes are provided mostly for mixed C and C++ programming. They enable the user to perform stream operations on files controlled by `FILE` structures, as defined in the C `stdio.h` header and implemented in the C run-time library. New C++ programs should avoid using these classes, as the facilities provided by **ifstream**, **ofstream** and **fstream** are more efficient.

A description of these classes can be found in section 6.6.

stdiobuf This class, which is derived from **streambuf**, supports the insertion and extraction operations via a **stdio.h** FILE structure.

stdiostream

This class is in fact derived directly from **ios**. It allows the user to perform insertions and extractions on **stdio.h** FILE structures, and to convert sequences of characters using the standard stream facilities.

6.1.2.6 Initialising the Stream Package

Iostream_init

The constructor function of this special class initialises the stream package's standard streams. The **iostream.h** header includes a declaration of a static member of this class, so the class constructor is called once each time the header is included, although the actual intialisation is only done once. In this way, the standard streams are always initialised before they are used.

Iostream_init has no public members, and the user should not normally be concerned with it. In some cases, however, global constructors may need to call the **Iostream_init** constructor explicitly, in order to ensure that the predefined streams have already been initialised correctly.

6.1.3 Predefined Streams

The following streams are predefined. As we have seen above, the predefinitions are performed by the constructor of the **Iostream_init** class.

Stream `cin` is of class `istream_withassign`. The others are of class `ostream_withassign`.

`cin` — This stream is connected to the MS-DOS `stdin` stream.

`cout` — This stream is connected to the MS-DOS `stdout` stream.

`cerr` — This stream is connected to the MS-DOS `stderr` stream. Output through this stream is not fully buffered, but only unit-buffered. This means that characters are flushed after every insertion. For more information, see the discussion of `unitbuf` in section 6.3.1.1, and section 6.2.4.1.

`clog` — This stream is also connected to the `MS-DOS` `stderr` stream, but unlike `cerr` its output is fully buffered.

The streams `cin`, `cerr` and `clog` are tied to `cout` so that any use of these will cause `cout` to be flushed. The performance of programs which copy from `cin` to `cout` may sometimes be improved by breaking the tie between `cin` and `cout` and doing explicit flushes of `cout`. See the discussion of the `tie` function in section 6.2.2.3 for details.

6.1.4 Header Files

Definitions for the stream package are held in a number of header files. Details will be found in the sections devoted to the various parts of the package. In the meantime, here is a summary of the contents of each header.

`iostream.h` — This header should be included in every program module which uses the stream package. It declares all the base classes and core stream classes described above, as well as the predefined streams.

fstream.h Declarations of all classes needed for input/output operations on files. Includes **iostream.h**.

strstream.h
Declarations of classes needed for operations on character arrays. Includes **iostream.h**.

stdiostream.h
Declarations of classes needed for operations on **stdio.h** FILE structures. Includes **iostream.h** and **stdio.h**.

iomanip.h Declarations of parameterised manipulators, as well as certain macros which help users who wish to create their own manipulators.

stream.h This header exists for compatibility with the earlier stream package. It includes **iostream.h**, **stdio.h**, and some other headers, and it declares some obsolete functions, enumerations, and variables. Some members of **streambuf** and **ios**, which are not discussed in this chapter, are present only for backward compatibility with the stream package.

6.2 Stream Input and Output

In this section, we shall discuss the base and core stream classes and their facilities, and in particular, unformatted input and output. Formatted input and output are discussed in section 6.3.

Section 6.2.1 deals with the constructor and assignment operations for these streams. Section 6.2.2 discusses features which are common to both input and output, while sections 6.2.3 and 6.2.4 discuss input and output respectively.

It is worth bearing in mind that although some of these classes will often be used by programs directly, some of the facilities described here are provided to support the classes which are derived from them.

Only the header **iostream.h** is required to use these facilities.

6.2.1 Constructors and Assignment

Note that these functions will not often be used directly by user programs. The core classes are most frequently used to access the MS-DOS standard streams, and these are predefined. Sometimes, however, a program may need to construct a stream which uses a predefined or existing **streambuf**, in which case these functions will be needed.

The old stream package allowed copying of streams; the current package does not. However, objects of the **istream_withassign**, **ostream_withassign** and **iostream_withassign** classes can be assigned to. These assignments actually associate the assigned stream with the other stream's **streambuf**. Old code which uses stream assignments can usually be rewritten to use these classes, or alternatively to use pointers to streams. The standard streams **cin**, **cout**, **cerr**, and **clog** are members of "withassign" classes, so they can be assigned to. For example:

```
cin = inputfstream;
```

If **inputfstream** is an object of class **ifstream**, the effect of this would be to associate **cin** with **inputfstream**'s **streambuf**, so that subsequent input through **cin** would come from the file controlled by **inputfstream**.

The old stream package had a constructor that took a stdio.h FILE* argument. This is no longer supported, and is not declared even as an obsolete form, in order to avoid having iostream.h depend on stdio.h. Users who need to access stdio.h FILE variables using the C++ stream library should use the facilities described in section 6.6.

6.2.1.1 Input Stream Classes

istream	istream constructor

```
istream(streambuf* sb);
```

This constructor associates the buffer **sb** with the **istream** and initialises the **istream**'s state variables.

istream_withassign	istream_withassign constructor

```
istream_withassign();
```

Constructs a stream but does no initialisation.

operator: =	assignment to an istream_withassign

```
istream_withassign& operator=(streambuf* sb);
```

Initialises the entire state of the assigned stream and associates **sb** with it.

operator: =	assignment to an istream_withassign

```
istream_withassign& operator=(istream& ins);
```

Initialises the entire state of the assigned stream and associates with it the buffer currently associated with **ins** (that is, **ins->rdbuf()**).

6.2.1.2 Output Stream Classes

ostream	ostream constructor

```
ostream(streambuf* sb);
```

This constructor associates the buffer sb with the ostream and initialises the ostream's state variables.

ostream_withassign	ostream_withassign constructor

```
ostream_withassign();
```

Constructs a stream but does no initialisation. This allows a file static variable of this type (cout, for example) to be used before it is constructed, provided it is assigned to first.

operator: =	assignment to an ostream_withassign

```
ostream_withassign& operator=(streambuf* sb);
```

Initialises the entire state of the assigned stream and associates sb with it.

operator: =	assignment to an ostream_withassign

```
ostream_withassign& operator=(ostream& outs);
```

Initialises the entire state of the assigned stream and associates with it the buffer currently associated with outs (that is, outs->rdbuf()).

6.2.1.3 Class ios

The `ios` class is the base from which all the stream classes are derived (see section 6.1.2.1). The information in this section will only needed by users who are building their own stream classes.

ios	**constructor for the `ios` class**

```
ios(streambuf* sb);
```

The `streambuf sb` becomes the streambuf associated with the constructed `ios`. If `sb` is null, the effect is undefined.

ios	**dummy constructor**

```
ios();
```

The `ios` class is used as a virtual base class for derived classes with multiple inheritance. For this reason, we need a constructor with no parameters. This constructor is declared protected, and performs no initialisation.

init	**initialise `ios` object**

```
void init(streambuf* sb);
```

When the `ios` class is used as a virtual base class (see above), no initialisation can be performed by the constructor. Accordingly `ios` includes this function as a protected member, which derived classes can use to initialise an `ios`.

ios	dummy constructor

```
ios(ios& iosr);
```

Copying of `ios` objects is not in general well-defined. This constructor with an `ios&` parameter is therefore declared private, but never defined. As a result, the compiler will flag any use of such a constructor as an error.

operator: =	assignment of ios object

```
void operator=(ios&);
```

Copying of `ios` objects is not in general well-defined. The assignment operator is therefore declared private, but never defined. As a result, the compiler will flag any attempt to assign a value to an `ios` object as an error.

6.2.2 Input and Output

This section describes facilities of the core classes which are common to unformatted input and output. These facilities are supported by members of the `ios` class, from which the other stream classes are derived.

Streams have a number of state variables, which initialised by their constructors, as described above. Amongst these are the *error state*, which is discussed in section 6.2.2.1 below, and the formatting state, which is discussed in detail in section 6.3.

6.2.2.1 Error Handling

A stream has an internal error state which is a collection of bits. These bits are referred to by **enum** values defined as part of the **ios** class, and so will normally need to be referenced with a scope qualifier, as shown below. They are:

ios::goodbit
: Despite its name, this refers to a state in which no error bits are set.

ios::eofbit
: Normally this bit is set when an end-of-file has been encountered during an extraction.

ios::failbit
: This bit indicates that some extraction or conversion has failed, but that the stream is still usable. In other words, once the **failbit** is cleared, input/output on this stream can usually continue.

ios::badbit
: This usually indicates that some operation on the stream's associated **streambuf** has resulted in a severe error, from which recovery is probably impossible.

Functions

The following functions are members of the **ios** class.

rdstate	return error state

```
int rdstate();
```

This function returns the stream's current error state.

clear	set error state

```
void clear(int i);
```

Stores `i` as the error state. The default value for `i` is 0. If `i` is zero, all bits are cleared. To set a bit without clearing previously set bits you need to read the existing value first; for example:

```
inputstream.clear(ios::badbit|inputstream.rdstate());
```

good	test for no errors

```
int good();
```

Returns non-zero if the error state has no bits set, zero otherwise.

eof	test for end-of-file

```
int eof();
```

Returns non-zero if `eofbit` is set in the error state, zero otherwise.

fail	test for error

```
int fail();
```

Returns non-zero if either `badbit` or `failbit` is set in the error state, zero otherwise. Note that if this function returns a non-zero value, it may be necessary to test separately for `badbit`.

bad	test for unrecoverable error

```
int bad();
```

Returns non-zero if `badbit` is set in the error state, zero otherwise.

Operators

The `ios` class includes the following two operators, which allow the error state of a stream to be checked conveniently.

operator: `void *`	test for no error

```
operator void*();
```

This operator converts a stream to a pointer so that it be compared to zero. (This pointer is not meant to be used). The conversion will return 0 if `failbit` or `badbit` is set in the error state, and will return a pointer value otherwise. This allows you to test the error state of a stream like this:

```
if ( cin >> x ) {
   // processing for successful completion
}
```

operator: !	test for error

```
int operator!();
```

The ! operator returns non-zero if `failbit` or `badbit` is set in the error state. This allows you test the error state like this:

```
if ( !cout ) {
   // processing for error condition
}
```

6.2.2.2 Positioning Streams

A stream can be thought of as a sequence of characters over which move one or two pointers. One pointer identifies the place at which characters can be fetched from the stream (the *get pointer*), and the other the place at which they may be stored (the *put pointer*).

Different classes of streams treat these pointers in different ways. Some, which are restricted to input or output only, have only one pointer. Others, such as those discussed in section 6.5, have two independent pointers, while others, such as the file streams discussed in section 6.4, have two pointers which always point to the same character.

Streams may be positioned by moving these pointers. Functions for doing this are discussed in sections 6.2.3.3 and 6.2.4.3, and are available for all classes of stream, even though some of these cannot in fact be positioned.

The differences in the handling of positioning for the various classes of streams are made not in the stream classes themselves, but in the corresponding buffer classes, all of which are derived from `streambuf` (see section 6.1.2.1). Derived buffer classes may provide their own versions of `streambuf`'s virtual `seekoff` and `seekpos` functions, which are then used by the stream positioning functions. Descriptions of `seekoff` and `seekpos` may be found in section 6.7, and the corresponding versions for other buffer classes are described in the appropriate sections. Most programmers, however, will only need to use the stream positioning functions described in sections 6.2.3.3 and 6.2.4.3 below.

The `ios` class includes definitions of the `enum seek_dir`, which is used to specify base locations from which to measure offsets as parameters to the stream and buffer positioning functions. This includes the following values:

`ios::beg` The beginning of the stream.

`ios::cur` The current position.

`ios::end` The end of the stream.

In addition, `iostream.h` defines two types used with these functions:

`streampos` A stream position. The programmer should not try to manipulate `streampos` values, using arithmetic operations, for example, but should treat them as opaque. Two particular values have special meanings:

- `streampos(0)` The beginning of the file.
- `streampos(EOF)` Used as an error indication.

`streamoff` An signed value used to express byte offsets from one of the base locations listed above.

6.2.2.3 Other Members

The class `ios` also includes the following function members.

rdbuf	pointer to streambuf

```
streambuf* rdbuf();
```

This function returns a pointer to the `streambuf` which was associated with the stream when it was constructed.

sync_with_stdio	synchronise standard streams

```
static void sync_with_stdio();
```

This function exists to solve problems which arise with the standard MS-DOS streams when input/output using the C++ stream package is mixed with standard C `stdio.h` input/output.

The first time it is called it will reset the standard streams (`cin`, `cout`, `cerr`, `clog`) to be streams using `stdiobuf`-class buffers (see section 6.6). After that, input and output on `stdin`, `stdout` and `stderr` using these streams may be mixed with input and output using the corresponding `FILE` structures, and will be properly synchronised. In addition, `sync_with_stdio` makes `cout` and `cerr` unit buffered. See section 6.2.4.1 and the discussion of `unitbuf` in section 6.3.1.1.

Invoking `sync_with_stdio` degrades the performance of input/output on the standard streams. The extent of this degradation depends on the length of the strings being inserted, with shorter strings performing worst.

The `sync_with_stdio` function is acknowledged to be an inelegant solution to this problem. The old stream package performed in this way by default, but with the current package unbuffered `stdiobufs` are regarded as too inefficient for this to continue. The function will only be needed with mixed C and C++ programs, and in general should be avoided.

`tie`	set tie variable

```
ostream* tie(ostream* osp);
```

The tie variable is the means by which different streams synchronise their operations. The tie variable is either null, or it points to an output stream. When a stream is about to fetch or a store characters, it flushes the stream which its tie variable points to, if any.

This function sets the stream's tie variable to osp, and returns its previous value.

By default, `cin` is tied initially to `cout` so that attempts to get more characters from standard input result in flushing standard output.

Additionally, `cerr` and `clog` are tied to `cout` by default. For other streams, the tie variable is set to zero by default.

tie	return value of tie variable

```
ostream * tie();
```

Returns the current value of the tie variable.

6.2.3 Input

This section discusses the facilities supported by the `istream` class and classes which are derived from it. Only unformatted operations are described here; for formatted input functions, see section 6.3.

6.2.3.1 Input Prefix Function

ipfx	input prefix function

```
int ipfx(int need);
```

This function is called by input functions before doing any transfer. Formatted input functions call `ipfx` with `need==0`, while unformatted input functions call it with `need==1`.

If necessary, the stream which is tied to this one (if any) is flushed (see the description of the `tie` function in section 6.2.2.3). This flushing is considered necessary if either `need==0` or if there are fewer than `need` characters immediately available.

After this, if the `ios::skipws` formatting flag is set (see section 6.3.1.1) and `need` is zero, leading whitespace characters are extracted from the stream and discarded.

If, on entry to **ipfx**, the stream's error state is non-zero, the function returns zero immediately. It also returns zero if an error occurs while skipping whitespace. Otherwise it returns non-zero.

6.2.3.2 Unformatted Input Functions

These functions call **ipfx(1)** (see section 6.2.3.1 above) and proceed only if it returns non-zero.

get	extract characters

```
istream& get(char* ptr, int lim, char delim);
istream& get(unsigned char* ptr ,int lim,
        char delim);
```

Extracts characters and stores them in the byte array beginning at **ptr** and extending for **len** bytes. Extraction stops when **delim** is encountered (**delim** is left in the stream and not stored), when the stream has no more characters, or when the array has only one byte left. The function always stores a terminating null, even if it does not extract any characters from the stream because of its error status. The error flag **ios::failbit** is set only if **get** encounters an end-of-file before it stores any characters.

The default value for **delim** is the newline character.

get	extract a single character

```
istream& get(unsigned char& c);
istream& get(char& c);
```

Extracts a single character and stores it in **c**.

get	extract a single character

```
int get();
```

Extracts a character and returns it. EOF is returned if extraction encounters an end-of-file. The error flag `ios::failbit` is never set.

get	extract characters, store in **streambuf**

```
istream& get(streambuf& sb, char delim);
```

This version of get extracts characters from the stream and stores them into `sb`. It stops if it encounters an end-of-file, if a store into `sb` fails or if it encounters `delim` (which it leaves in the stream). The error flag `ios::failbit` is set if it stops because the store into `sb` fails.

getline	extract characters and terminator

```
istream& getline(char* ptr, int lim, char delim);
istream& getline(unsigned char* ptr, int lim,
         char delim);
```

Extracts characters and stores them in the byte array beginning at `ptr` and extending for `len` bytes. Extraction stops when `delim` is encountered (`delim` is extracted from the stream and stored), when the stream has no more characters, or when the array is full. If `delim` occurs when exactly `len` characters have been extracted, termination is treated as being due to the array being filled, and this `delim` is left in the stream. The error flag `ios::failbit` is set only if the function encounters an end-of-file before it stores any characters.

The default value for `delim` is the newline character.

ignore	skip characters

```
istream& ignore(int n, int delim);
```

Extracts and throws away up to n characters. Extraction stops prematurely if `delim` is extracted or end of file is reached. If `delim` is `EOF` it can never cause termination.

The default value for n is 1. For `delim`, the default value is `EOF`.

read	extract characters

```
istream& read(char* ptr, int n);
istream& read(unsigned char* ptr, int n);
```

Extracts n characters and stores them in the array beginning at `ptr`. If end of file is reached before n characters have been extracted, `read` stores whatever it can extract and sets the error flag `ios::failbit`. The number of characters extracted can be determined via gcount (see section 6.2.3.4 below).

6.2.3.3 Positioning Functions

These functions are members of `istream`. For a discussion of stream positioning, see section 6.2.2.2. Note that the predefined streams do not support positioning.

seekg	move the get pointer

```
istream& seekg(streampos pos);
```

This function moves the get pointer of the buffer associated with this stream to the position pos.

seekg	move the get pointer

```
istream& seekg(streamoff off,
      ios::seek_dir dir);
```

This function moves the get pointer of the buffer associated with this stream. The `dir` parameter is one of the location bases `beg`, `cur` or `end` discussed in section 6.2.2.2; `off` is a byte offset from this location.

tellg	current position of get pointer

```
streampos tellg();
```

This function returns the current position of the get pointer of the buffer associated with this stream.

6.2.3.4 Other Members

The following functions are also members of `istream`.

gcount	number of characters fetched

```
int gcount();
```

Returns the number of characters fetched by the last unformatted input function. Note that formatted input functions may call unformatted input functions and thereby reset this number.

peek	look ahead

```
int peek();
```

This function calls the input prefix function `ipfx` with a parameter value `1`. If that call returns zero or if the stream is at end-of-file, it returns `EOF`. Otherwise it returns the next character without extracting it, that is, without moving the get pointer.

putback	back up stream

```
istream& putback(char c);
```

This function attempts to back up the buffer associated with this stream. The parameter `c` must be the character before the get pointer. (Unless other activity is modifying the buffer, this is the last character fetched from the stream.) If it is not, the effect is undefined.

The function may fail (and set the error state). Although it is a member of `istream`, `putback` never extracts characters, so it does not call `ipfx`. It will, however, return without doing anything if the error state is non-zero.

sync	synchronise stream and source

```
int sync();
```

This function ensures that the internal data structures and the external source of characters are consistent with each other. The function works by calling the buffer's `sync` function. This is a virtual function, so the details depend upon the derived buffer class.

The function returns `EOF` to indicate errors.

6.2.4 Output

This section discusses the facilities supported by the `ostream` class and classes which are derived from it. Only unformatted operations are described here; for formatted output functions, see section 6.3.

6.2.4.1 Output Prefix and Suffix Functions

`opfx`	output prefix function

```
int opfx();
```

This function is called by output functions before doing any transfer. It flushes the stream which is tied to this one, if any; see the description of the `tie` function in section 6.2.2.3.

If, on entry to `opfx`, the stream's error state is non-zero, the function returns zero immediately. Otherwise it returns non-zero.

`osfx`	output suffix function

```
void osfx();
```

This function is called by every formatted output function (inserter) after performing the transfer and before returning to the user.

If the formatting flag `ios::unitbuf` is set, `osfx` flushes the stream. If the formatting flag `ios::stdio` is set, `osfx` flushes `stdout` and `stderr`. See section 6.3.1.1 for a discussion of these flags.

The output suffix function is called by all predefined inserters, and should be called by user-written inserters as well, if they manipulate the associated buffer directly. It is not called by the unformatted output functions.

6.2.4.2 Unformatted Output Functions

These functions are members of the `ostream` class.

put	output one character

```
ostream& put(char c);
```

This function stores c in the associated buffer. It sets the error state of the stream if the operation fails.

write	write characters

```
ostream& write(const char *s, int n);
```

Stores the n characters starting at s in the associated buffer. These characters may include zeros; that is, s is not treated like a zero-terminated string.

flush	flush the stream

```
ostream& flush();
```

When characters are stored in a buffer, they are not necessarily sent to the character sink at once. For example, if the sink is an external file, characters are not always written out to the file at once, but may be held temporarily in memory.

When `flush` is invoked, any characters which have been stored in the buffer but are still waiting to be sent to the sink are sent there at once. It does this by calling the buffer's sync function. This is a virtual function, so the details depend upon the derived buffer class.

6.2.4.3 Positioning Functions

These functions are members of `ostream`. For a discussion of stream positioning, see section 6.2.2.2. Note that the predefined streams do not support positioning.

`seekp`	move the put pointer

```
ostream& seekp(streampos pos);
```

This function moves the put pointer of the buffer associated with this stream to the position `pos`.

`seekp`	move the put pointer

```
ostream& seekp(streamoff off,
        ios::seek_dir dir);
```

This function moves the put pointer of the buffer associated with this stream. The `dir` parameter is one of the location bases `beg`, `cur` or `end` discussed in section 6.2.2.2; `off` is a byte offset from this location.

`tellp`	current position of put pointer

```
streampos tellp();
```

This function returns the current position of the put pointer of the buffer associated with this stream.

6.3 Formatted Input and Output

The section discusses formatted input and output operations on streams. Unformatted input and output are discussed in section 6.2.

6.3.1 The Formatting State

A stream has a formatting state which decides the details of the way input and output formatting are done. The formatting state has the following components:

- A number of *formatting flags*.
- The *fill character*.
- The *precision variable*.
- The *width variable*.

The formatting flags and the functions to control them are described next, in section 6.3.1.1. The other components are discussed in section 6.3.1.2 below.

The formatting state affects only formatted input and output operations. For other operations the format state has no particular effect and its components may be set and examined arbitrarily by user code.

6.3.1.1 Formatting Flags

The following flags for specifying format states are defined in the `ios` class, and so will usually need to be specified with an `ios::` scope qualifier.

skipws If **skipws** is set, whitespace will be skipped on input. This applies to scalar extractions. See also section 6.2.3.1.

Zero width fields are considered a bad format by the extractors, so if the next character is whitespace and **skipws** is not set, the arithmetic extractors will signal an error.

left, right, internal
When a value is converted for output, the resulting character string may be shorter than the width expected. In this case, the string padded with fill characters. These flags control the way in which this is done, as follows:

- When **left** is set, the value is left-adjusted, that is, the fill characters are added after the value.
- When **right** is set, the value is right-adjusted, that is the fill character is added before the value.
- When **internal** is set, the fill character is added after any leading sign or base indication, but before the value.

Right-adjustment is the default if none of these flags is set. The fill character is defined by the **fill** function, and the width of padding is defined by the **width** function; see section 6.3.1.2.

The current values of these flags are held in the static member **ios::adjustfield**.

dec, oct, hex
These flags control the current conversion base. If **dec** is set, input and output are performed using base 10 (decimal); similarly, **hex** specifies hexadecimal and **oct** octal conversion. If none of these is set, output is in

decimal, but input is interpreted according to the C++ lexical conventions for integral constants.

The manipulators **hex**, **dec**, and **oct** can also be used to set the conversion base; see section 6.3.4.1 below. The current values of these flags are held in the static member **ios::basefield**.

showbase If **showbase** is set, output values will be converted to an external form that can be read according to the C++ lexical conventions for integral constants. **showbase** is unset by default.

showpos If **showpos** is set, then a '' will be inserted into a decimal conversion of a positive integral value.

uppercase If **uppercase** is set, then an uppercase '**X**' will be used for hexadecimal conversion when **showbase** is set, or an uppercase '**E**' will be used to print floating point numbers in scientific notation.

showpoint If **showpoint** is set, trailing zeros and decimal points appear in the result of a floating point conversion.

scientific, fixed
These flags control the format in which floating-point values are output.

- If **scientific** is set, the value is converted using scientific notation, where there is one digit before the decimal point and the number of digits after it is equal to the current precision. An uppercase '**E**' will introduce the exponent if **uppercase** is set, a lowercase '**e**' will appear otherwise.
- If **fixed** is set, the value is converted to decimal notation. The number of digits after the decimal point is equal to the current precision.

- If neither is set, then the value will be converted using either notation, depending on the value; scientific notation will be used only if the exponent resulting from the conversion is less than −4 or greater than the current precision. If **showpoint** is not set, trailing zeroes are removed from the result and a decimal point appears only if it is followed by a digit.

The precision is defined by the **precision** function; see section 6.3.1.2 below. The current value of these flags is held in the static member **ios::floatfield**.

unitbuf When set, a flush is performed by the output suffix function **ostream::osfx** after each insertion. Unit buffering provides a compromise between buffered output and unbuffered output. Performance is better under unit buffering than unbuffered output, which makes a system call for each character output. Unit buffering makes a system call for each insertion operation, and does not require the user to call **ostream::flush**. See section 6.2.4.1.

stdio When set, **stdout** and **stderr** are flushed by the output suffix function **ostream::osfx** after each insertion. See section 6.2.4.1.

Functions

The following functions are provided to manipulate the flags discussed above. They are members of the **ios** class.

flags	current format flags

```
long flags();
```

Returns the current values of the format flags.

flags	specify format flags

```
long flags(long f);
```

Sets the format flags to the values specified in `f` and returns the previous settings. All the previous flag settings are lost.

setf	set format flags

```
long setf(long bits);
```

Turns on the format flags marked in `bits` and returns the previous settings. Flags which were set before are not changed. A parameterised manipulator, `setiosflags`, performs the same function; see section 6.3.4.2.

setf	set flags in field

```
long setf(long bits, long field);
```

All the flags in the member `field` are clear, and then set to the values specified in `bits`. For example, to change the conversion base to be `hex`, one could write:

```
s.setf(ios::hex,ios::basefield)
```

As we saw in the discussion of the conversion flags above, `ios::basefield` holds all the conversion base bits. These will be cleared and replaced by `ios::hex`.

Note that `s.setf(0,field)` will clear all the bits in `field`. The parameterised manipulator `resetiosflags` has the same effect; see section 6.3.4.2.

unsetf	unset format flags

```
long unsetf(long bits);
```

Unsets the flags set in **bits** and returns the previous settings.

6.3.1.2 Other Formatting Variables

The functions described here are all members of the **ios** class.

Width

The width variable specifies the minimum width of the character string to be output by the next call on an inserter function. If the field width is zero (the default), inserters will insert only as many characters as necessary to represent the value being inserted. If the field width is non-zero, the inserters will always insert at least that many characters. If the result of performing the conversion is shorter than this value, the string will be padded with the fill character in the way specified by the **left**, **right** and **internal** flags.

The numeric inserters never truncate values, even if the value being inserted will not fit in the specified width. There is no direct way to specify a maximum number of characters.

The width is reset to the default (zero) after each insertion or extraction.

width	set field width

```
int width(int w);
```

Sets the width variable to **w** and returns the previous value. The parameterised manipulator **setw** is also available for setting the width; see section 6.3.4.2.

`width`	field width

```
int width();
```

Returns the width variable.

The Fill Character

The fill character is used to pad output strings which are shorter than the current field width. The default fill character is a space. The way in which padding is done is defined by the `left`, `right` and `internal` flags; see above.

`fill`	set the fill character

```
char fill(char c);
```

Sets the fill character to `c` and returns the previous value. `c` will be used as the padding character, if one is necessary (see `width()`, below). The parameterised manipulator `setfill` is also available for setting the fill character; see section 6.3.4.2.

`fill`	fill character

```
char fill();
```

Returns the fill character.

Precision

This variable defines the number of significant digits output when a floating-point value is inserted. Details of how it is used can be found above, in the discussion of the `scientific` and `fixed` flags. The default value for precision is 6.

precision	set the precision

```
int precision(int i);
```

Sets the precision format state variable to **i** and returns the previous value. The parameterised manipulator **setprecision** may also be used for this purpose; see section 6.3.4.2.

precision	current precision

```
int precision();
```

Returns the current precision.

6.3.2 Extraction: The >> Operator

The **>>** operator is overloaded by the **istream** class to perform formatted extractions and certain other functions.

Note that there is no overflow detection on the conversion of integers.

operator: >>	extract

```
istream& operator>>(type x);
```

The operator first calls the input prefix function, **ipfx**, with a parameter 0 (see section 6.2.3.1). If that returns non-zero, it then extracts characters from the stream, converts them according to the type of **x** and stores the converted value in **x**.

A reference to the stream is returned, so that expressions of the following sort are allowed:

```
cin >> a >> b;
```

Errors are indicated by setting the stream's error state. The error flag `ios::failbit` is set if the characters extracted were not a representation of the required type. The error flag `ios::badbit` is set if attempts to extract characters failed.

The details of the conversion performed depend on the values of the stream's format state flags and variables (see section 6.3.1 above) and the type of `x`. Extractions which use the width variable reset it to 0, but apart from this, the extraction operators do not change the value of the stream's format state.

Extractors are defined for the following types, with conversion rules described below.

`char*, unsigned char*`
: Characters are stored in the array pointed at by `x` until a whitespace character is found. The terminating whitespace is left in the stream. If the width variable is non-zero it is taken to be the size of the array, and the maximum number of characters extracted will be one less than the width. A terminating null character (0) is always stored (even when nothing else is done because of the stream'a error state). The width variable is reset to 0.

`char&, unsigned char&`
: A single character is extracted and stored in `x`.

`short&, unsigned short&, int&, unsigned int&, long&, unsigned lon`
: Characters are extracted and converted to an integral value according to the conversion specified in the stream's format flags. The converted value is stored in `x`.

  The first character may be a sign ('+' or '-'). After that, if `ios::oct`, `ios::dec`, or `ios::hex` is set, the conversion is octal, decimal, or hexadecimal, respectively. Conversion is terminated by the first non-digit, which is left in the stream.

If none of the conversion base format flags is set, then the number is interpreted according to C++ lexical conventions. That is, if the first characters (after the optional sign) are `0x` or `0X` a hexadecimal conversion is performed, if the first character is `0`, an octal conversion is performed, and in all other cases a decimal conversion is performed.

The error flag `ios::failbit` is set if there are no characters available which correspond with the expected format.

`float&`, `double&`
Converts the characters according to C++ syntax for a `float` or `double`, and stores the result in `x`. The error flag `ios::failbit` is set if there are no digits available or if they do not start with a well-formed floating-point number.

`&streambuf` Extracts characters from the stream and inserts them into the `streambuf`. Extraction stops when `EOF` is reached.

manipulator Syntactically, the use of a manipulator resembles an extractor operation. For example:

```
cin >> oct;
```

This does not, however, convert a sequence of characters and store them is `oct`. Instead, it sets `cin`'s conversion base to `ios::oct`. Other manipulators perform other operations. Further discussion of manipulators can be found in section 6.3.4.

In addition, users can write extractor functions for their own classes, which can then be invoked using the `>>` operator and the same syntax.

6.3.3 Insertion: The << Operator

The << operator is overloaded by the `ostream` class to perform formatted insertions and certain other functions.

operator: <<	insert

```
ostream& operator<<(type x);
```

The operator first calls the output prefix function, `opfx` (see section 6.2.4.1. If that returns non-zero, it converts `x` into a sequence of characters called the *representation*. Next, padding is performed as specified by the width formatting variable and the `left`, `right` and `internal` flags (see section 6.3.1). The representation is then inserted into the stream's associated buffer. Finally, the operator calls the output suffix function (see section 6.2.4.1).

A reference to the stream is returned, so that expressions of the following sort are allowed:

```
cout << a << b;
```

Errors are indicated by setting the stream's error state.

The details of the conversion performed depend on the values of the stream's format state flags and variables (see section 6.3.1 above) and the type of `x`. Except that insertions that use width reset it to 0, the insertion operators do not change the value of the stream's format state.

Inserters are defined for the following types, with conversion rules described below.

`char*` The representation is the sequence of characters up to (but not including) the terminating null of the string `x` points at.

integral types except `char` and `unsigned char`
Decimal, octal or hexadecimal conversion is performed,

depending on which of the formatting flags `ios::dec`, `ios::oct` or `ios::hex` is set. If none of them is set, decimal conversion is performed.

If `x` is zero, the representation is '0'. If `x` is negative, decimal conversion converts it to a minus sign '-' followed by decimal digits. If `x` is positive and `ios::showpos` is set, decimal conversion converts it to a plus sign (+) followed by decimal digits. The other conversions treat all values as unsigned. If `ios::showbase` is set in `ios`'s format flags, the hexadecimal representation contains "0x" before the hexadecimal digits, or "0X" if `ios::uppercase` is set. If `ios::showbase` is set, the octal representation contains a leading 0.

`void*` Pointers are converted to integral values and then converted to hexadecimal numbers as if `ios::showbase` were set.

`float, double`

The arguments are converted according to the current value of the precision and width formatting variables and the format flags `ios::scientific`, `ios::fixed`, and `ios::uppercase`. See section 6.3.1 for details.

`char, unsigned char`

The character is output unchanged.

`&streambuf` Characters are fetched from the specified `streambuf` and inserted into the stream's associated buffer. Insertion stops when no more characters can be fetched. No padding is performed.

manipulator Syntactically, the use of a manipulator resembles an inserter operation. For example:

```
cout << oct;
```

This does not, however, convert the value of `oct` to a sequence of characters and store them in `cout`. Instead, it sets `cout`'s conversion base to `ios::oct`. Other manipulators perform other operations. Further discussion of manipulators can be found in section 6.3.4.

In addition, users can write inserter functions for their own classes, which can then be invoked using the << operator and the same syntax.

6.3.4 Manipulators

As we have seen, using a manipulator is syntactically similar to an insertion or extraction operation. However, it is in fact a function call. For example:

```
cout << flush;
cin >> ws;
```

These are equivalent to the following:

```
flush(cout);
ws(cin);
```

The manipulators which are provided as part of the stream package can be conveniently divided into simple manipulators, which are used without parameters, and parameterised manipulators. As we shall see, users may also build their own manipulators.

6.3.4.1 Simple Manipulators

The following manipulators are all functions which have as their single parameter an `ios&`, an `istream&` or an `ostream&`, and return their argument. In the descriptions below, `sr` is a `ios&`.

manipulator: **dec**	set decimal

```
sr<<dec;
sr>>dec;
```

These set the conversion base format flag of **sr** to 10, so that future conversions use decimal representations.

manipulator: **hex**	set hexadecimal

```
sr<<hex;
sr>>hex;
```

These set the conversion base format flag of **sr** to 16, so that future conversions use hexadecimal representation.

manipulator: **oct**	set octal

```
sr<<oct;
sr>>oct;
```

These set the conversion base format flag of **sr** to 8, so that future conversions use octal representation.

manipulator: **ws**	extract whitespace

```
sr>>ws;
```

This manipulator extracts whitespace characters from **sr** and discards them.

manipulator: **endl**	end of line

```
sr<<endl;
```

Ends a line by inserting a newline character into `sr` and flushing the stream.

manipulator: **ends**	end of string

```
sr<<ends;
```

Ends a string by inserting a null (\0) character into `sr`.

manipulator: **flush**	flush stream

```
sr<<flush;
```

This manipulator flushes `sr`. See section 6.2.4.2, where the `flush` function is discussed in more detail.

6.3.4.2 Parameterised Manipulators

The following manipulators are declared in the header file `manip.h`, which must be included in any program which uses them.

These manipulators all have to do with changing the format state of a stream; see section 6.3.1 for further details. In the descriptions below, `ostr` represents an `ostream` and `istr` represents an `istream`.

manipulator: **setw**	set width

```
 ostr<<setw(n);
istr>>setw(n);
```

Sets the width formatting variable of **ostr** or **istr** to the value of the **int** parameter n. This is the equivalent of a call to the **ios::width** function.

manipulator: **setfill**	set fill character

```
 ostr<<setfill(n);
istr>>setfill(n);
```

Sets the fill character of **ostr** or **istr** to the value of the **int** parameter n. This is the equivalent of a call to the **ios::fill** function.

manipulator: **setprecision**	set precision

```
 ostr<<setprecision(n);
istr>>setprecision(n);
```

Sets the precision formatting variable of **ostr** or **istr** to the value of the **int** parameter n. This is the rquivalent of a call to the **ios::precision** function.

manipulator: **setiosflags**	set formatting flags

```
 ostr<<setiosflags(bits);
istr>>setiosflags(bits);
```

Turns on the format flags in **ostr** or **istr**. The flags to turn on are specified in the long parameter bits. Flags which were set before are not changed. This is the equivalent of a call ios::setf(bits).

manipulator: **resetiosflags**	reset formatting flags

```
ostr<<resetiosflags(field);
istr>>resetiosflags(field);
```

Clears format flags in ostr or istr. The long parameter field specifies the field to reset. For example, the following would reset the conversion base of ostr to 0:

```
ostr<<resetioflagsios::basefield;
```

This is the equivalent of a call to ios::setf(0, field).

6.3.5 User Extensions

6.3.5.1 Formatting Flags

Class ios can be used as a base class for derived classes that require additional format flags or variables. The iostream library provides several functions to do this. The two static member functions ios::xalloc and ios::bitalloc allow several such classes to be used together without interference. See section 6.3.1.

These functions are all members of the ios class.

bitalloc	get unused formatting bit

```
static long bitalloc();
```

This function returns a long in which a single bit will be set. This bit is a previously-unused formatting flag. This allows users who need an additional flag to acquire one, and pass it as an argument to `ios::setf`, for example.

`xalloc`	allocate index to free words

```
static int xalloc();
```

This function returns a previously unused index into an array of words available for use as format state variables by derived classes.

`iword`	find user-defined word

```
long& iword(int i);
```

When `i` is an index allocated by `ios::xalloc`, `iword` returns a reference to the `i`th user-defined word.

`pword`	find user-defined word

```
void*& pword(int i);
```

When `i` is an index allocated by `ios::xalloc`, `pword` returns a reference to the `i`th user-defined word. This function is the same as `iword` except that it is typed differently.

6.3.5.2 Parameterised Manipulators

The header file `iomanip.h` supplies macro definitions which programmers can use to define new parameterised manipulators.

Ideally, the types relating to manipulators would be parameterised as "templates". The macros defined in `iomanip.h` are used to simulate templates. `IOMANIPdeclare(T)` declares the various classes and operators. (All code is declared inline so that no separate definitions are required.) Each of the other Ts is used to construct the real names and therefore must be a single identifier. Each of the other macros also requires an identifier and expands to a name.

In the following descriptions, assume:

`t` is a T, or type name.
`s` is an `ios`.
`i` is an `istream`.
`o` is an `ostream`.
`io` is an `iostream`.
`f` is an `ios& (*)(ios&)`.
`if` is an `istream& (*)(istream&)`.
`of` is an `ostream& (*)(ostream&)`.
`iof` is an `iostream& (*)(iostream&)`.

```
s<<SMANIP(T)(f,t)
s>>SMANIP(T)(f,t)
s<<SAPP(T)(f)(t)
s>>SAPP(T)(f)(t)
```

Returns `f(s,t)`, where `s` is the left operand of the insertion or extractor operator (i.e., `s`, `i`, `o`, or `io`).

```
i>>IMANIP(T)(if,t)
i>>IAPP(T)(if)(t)
```

Returns `if(i,t)`.

```
o<<OMANIP(T)(of,t)
o<<OAPP(T)(of)(t)
```

Returns `of(o,t)`.

```
io<<IOMANIP(T)(iof,t)
io>>IOMANIP(T)(iof,t)
io<<IOAPP(T)(iof)(t)
io>>IOAPP(T)(iof)(t)
```

Returns `iof(io,t)`.

`iomanip.h` contains declarations of `IOMANIPdeclare(int)` and `IOMANIPdeclare(long)`.

Syntax errors will be reported if `IOMANIPdeclare(T)` occurs more than once in a file with the same T.

6.4 Operations on Files

This section describes the stream library's facilities for performing input/output on files.

Programs which use these facilities must include the header file `fstream.h`.

Four new classes are introduced to support file I/O.

Three of these are the stream classes `ifstream`, `ofstream` and `fstream`. They are derived respectively from the core classes `istream`, `ostream` and `iostream` and so support all the facilities described in sections 6.2 and 6.3. In addition, they include members for opening and closing files and other operations.

The fourth new class is a buffer class, `filebuf`, which is derived from `streambuf`. The buffers used by the file streams are of class `filebuf`, and most of the facilities provided by the stream classes make use of functions which are members of `filebuf`. Most users will not need to use `filebuf` members, but will use the stream classes instead; such users can disregard most of the discussion of `filebuf` in this section.

The filebuf class specialises **streambuf** to use a file as a source or sink of characters. Characters output by the program are in the end written out into a file, while the characters which the program needs for input are read from a file. The `filebuf` allows a file to be positioned, if this is possible. At least 4 characters of putback are guaranteed. When the file permits reading and writing, the `filebuf` permits both storing and fetching; unlike the C `stdio.h` functions, `filebuf` requires no special action between gets and puts.

A `filebuf` accesses the environment through a value called a *file descriptor*. When a `filebuf` is connected to a file descriptor, it (and its associated stream) is said to be open. Stream and filebuf members for opening files have a parameter for specifying a protection mode; under MS-DOS, this is disregarded.

A `filebuf` controls a buffer called the *reserve area* (see section 6.7). This is used for buffering transfers to and from the file. The reserve area may be specified explicitly by a constructor or by calling the `setbuf` function; if this is not done, one is allocated automatically. You can also make a `filebuf` unbuffered, in which case characters are transferred to and from the file one-by-one. The get and put pointers into the reserve area are conceptually tied together; they behave as a single pointer. Therefore, the descriptions below refer to a single get/put pointer.

6.4.1 Constructors

6.4.1.1 Stream Constructors

The constructors for the three stream classes are similar. In the descriptions below, "`STREAM`" stands for `ifstream`, `ofstream` or `fstream`. In practice, most users use `fstream` for all files.

`STREAM`	constructor for file streams

```
STREAM();
```

Constructs an unopened stream.

STREAM	constructor for file streams

```
STREAM(const char *name, int mode, int prot);
```

Constructs a stream and opens the file specified by `name` using the specified `mode` as the open mode. (Open modes are described in detail in the discussion of the **open** function in section 6.4.2 below.)

The `prot` parameter is included for compatibility with other systems, but is disregarded by this implementation.

If the open fails, the error state of the constructed stream is set to indicate failure.

STREAM	constructor for file streams

```
STREAM(int fd);
```

Constructs a stream connected to file descriptor `fd`, which must be already open. The `filebuf::fd` function can be used to access the file descriptor of an open stream. Notice that no test is made to check that the file descriptor supplied is in fact valid or open.

STREAM	constructor for file streams

```
STREAM(int fd, char *ptr, int len);
```

Constructs a stream connected to file descriptor `fd`, and, in addition, initialises the associated `filebuf` to use the `len` bytes at `ptr` as the reserve area. If `ptr` is `NULL` or `len` is `0`, the `filebuf` will be unbuffered.

6.4.1.2 Buffer Constructors

`filebuf`	`filebuf` constructor

```
filebuf();
```

Constructs an initially closed `filebuf`.

`filebuf`	`filebuf` constructor

```
filebuf(int fd);
```

Constructs a `filebuf` connected to file descriptor `fd`. The `filebuf::fd` function can be used to access the file descriptor of an open `filebuf`. Notice that no test is made to check that the file descriptor supplied is in fact valid or open.

`filebuf`	`filebuf` constructor

```
filebuf(int fd, char *ptr, int len);
```

Constructs a `filebuf` connected to file descriptor `fd` and initialised to use the reserve area starting at `ptr` and containing `len` bytes. If `ptr` is `NULL` or `len` is zero or less, the `filebuf` will be unbuffered.

6.4.2 Stream Operations

Each of the three stream classes have members which follow the descriptions given below.

6.4.2.1 Opening and Closing Streams

open	**open a stream**

```
void open(const char *name, int mode, int prot);
```

The function opens the file specified in `name` and connects the stream to it. It makes a call to the associated `filebuf`'s `open` member, and if this fails, `ios::failbit` is set in the stream's error state. This is also done if the file is already open.

The `prot` parameter is provided for compatibility with other systems, but is ignored in this implementation.

The `mode` parameter specifies the mode with which the file is to be opened. The `ios` class contains a definition of the `open_mode` `enum`, and its members can be used to specify this parameter. These members are bits which can be ORed together; as this OR operation produces an `int` value, the `mode` parameter is an `int`.

Note that if the file does not exist, and the mode bit `ios::nocreate` is not specified, an attempt will be made to create the file.

The meanings of the mode bits are as follows.

`ios::app` The stream is positioned at the end of file. Subsequent data written to the file are always added (appended) at the end of file. This mode bit implies `ios::out`.

`ios::ate` The stream is positioned at the end of file. This mode bit does not imply `ios::out`.

ios::in The file is opened for input. This bit is implied when you are constructing or opening an **ifstream**. For an **fstream** it indicates that input operations should be allowed if possible. It is legal to include **ios::in** in the modes of an **ostream** in which case it implies that the original file (if it exists) should not be truncated.

ios::out The file is opened for output. This bit is implied when you are constructing or opening an **ofstream**. For an **fstream** it indicates that output operations are to be allowed.

ios::trunc If the file already exists, its contents will be truncated (discarded). This mode is implied when **ios::out** is specified (including implicit specification for **ofstream**) and neither **ios::ate** nor **ios::app** is specified.

ios::nocreate
If the file does not already exist, the open will fail.

ios::noreplace
If the file already exists, the open will fail.

attach	attach stream to file descriptor

```
void attach(int fd);
```

Connects the stream to the file descriptor **fd**. If the stream is already connected to a file, **ios::failbit** in the stream's error state is set.

The **filebuf::fd** function can be used to access the file descriptor of an open stream. Notice that no test is made to check that the file descriptor supplied is in fact valid or open.

close	close a stream

```
void close();
```

Closes any associated `filebuf` and thereby breaks the connection between the stream and the file. The stream's error state is cleared except on failure. A failure occurs when the call to the associated `filebuf`'s `close` member fails.

6.4.2.2 Positioning Streams

The functions `seekp` and `tellp` are allowed for `ofstream` streams, and `seekg` and `tellg` are allowed for `ifstream`. All four are allowed for `fstream`. However, the two pointers are in fact tied together, and the same position is used for both fetching and storing characters.

Positioning is implemented by using the `filebuf` versions of the virtual functions `seekoff` and `seekpos`, which are discussed in section 6.4.3 below. See also section 6.2.2.2 for a general discussion of positioning.

6.4.2.3 Other Operations

rdbuf	access associated `filebuf`

```
filebuf *rdbuf();
```

Returns a pointer to the `filebuf` associated with the stream. `fstream::rdbuf` has the same meaning as `iostream::rdbuf` but is typed differently.

setbuf	create reserved area

```
void setbuf(char *ptr, int len);
```

Initialises the associated `filebuf` to use the `len` bytes at `ptr` as the reserve area. If `ptr` is `NULL` or `len` is `0`, the `filebuf` will be unbuffered. A failure occurs if the stream is open or the call to the associated `filebuf`'s `setbuf` member fails.

6.4.3 Buffer Operations

These functions are members of the `filebuf` class. As was noted above, most users will not need to use these functions, but will use the stream facilities instead.

6.4.3.1 Opening and Closing Buffers

open	open a buffer

```
filebuf *open(const char *name, int mode,
        int prot);
```

Opens the file specified by `name` and connects the `filebuf` to it. If the file does not already exist, an attempt is made to create it, unless `ios::nocreate` is specified in `mode`.

For a discussion of the values of the `mode` parameter, see the description of `fstream::open` in section 6.4.2. The `prot` parameter is provided for compatibility, but is ignored in this implementation.

Failure occurs if the `filebuf` is already open. The function normally returns the address of the `filebuf` but if an error occurs it returns `0`.

close	close a buffer

```
filebuf *close();
```

Flushes any waiting output, closes the file descriptor, and disconnects the **filebuf**. Unless an error occurs, the **filebuf**'s error state will be cleared. The function returns the address of the **filebuf** unless errors occur, in which case it returns 0. Even if errors occur, **close** leaves the file descriptor and the **filebuf** closed.

fd	return file descriptor

```
int fd();
```

Returns the file descriptor which the **filebuf** is connected to. If the **filebuf** is closed, EOF is returned.

attach	connect buffer to file descriptor

```
filebuf *attach(int fd);
```

Connects the **filebuf** to an open file descriptor, **fd**. The function normally returns the address of the **filebuf**, but returns 0 if the **filebuf** is already open.

Notice that no test is made to check that the file descriptor supplied is in fact valid or open.

is_open	check if buffer is open

```
int is_open();
```

Returns non-zero when the `filebuf` is connected to a file descriptor, and zero otherwise.

6.4.3.2 Positioning Buffers

For a general discussion of positioning, see section 6.2.2.2.

seekoff	position buffer by offset

```
streampos seekoff(streamoff off,ios::seek_dir dir,
      int mode);
```

Moves the get/put pointer as designated by `off` and `dir`. It may fail if the file that the `filebuf` is attached to does not support seeking, or if the attempted motion is otherwise invalid (such as attempting to seek to a position before the beginning of file).

The `off` parameter is interpreted as a count relative to the place in the file specified by `dir`; for a description of `dir`, see section 6.2.2.2. The `mode` parameter is ignored, as the two pointers are not independent. The function returns the new position, or `EOF` if a failure occurs. The position of the file after a failure is undefined. The `mode` parameter is ignored.

seekpos	position buffer

```
streampos seekpos(streampos pos, int mode);
```

Moves the file to a position `pos`. The `mode` parameter is ignored. The function normally returns `pos`, but on failure it returns `EOF`.

6.4.3.3 Other Operations

setbuf	create reserve area

```
streambuf *setbuf(char *ptr, int len);
```

Sets up the reserve area as **len** bytes beginning at **ptr**. If **ptr** is **NULL** or **len** is less than or equal to 0, the **filebuf** will be unbuffered. The function normally returns the address of **filebuf**. However, if the **filebuf** is open and a buffer has been allocated, no changes are made to the reserve area or to the buffering status, and **setbuf** returns 0.

sync	synchronise buffer with file

```
int sync();
```

Attempts to force the state of the get/put pointer of the **filebuf** to agree (be synchronised) with the state of the file to which it is connected. This means it may write characters to the file if some have been buffered for output or attempt to reposition (seek) the file if characters have been read and buffered for input.

Normally, **sync** returns 0, but it returns **EOF** if synchronisation is not possible.

Sometimes it is necessary to guarantee that certain characters are written together. To do this, the program should use **setbuf** (or a constructor) to guarantee that the reserve area is at least as large as the number of characters that must be written together. It can then call **sync**, then store the characters, then call **sync** again.

6.5 In-Store Operations

This section describes the stream library's facilities for performing in-store operations, that is, storing and fetching from arrays of bytes.

Programs which use these facilities must include the header file `strstream.h`.

Four new classes are introduced to support in-store operations.

Three of these are the stream classes: `istrstream`, `ostrstream` and `strstream`. They are derived respectively from the core classes `istream`, `ostream` and `iostream` and so support all the facilities described in sections 6.2 and 6.3. In addition, they include members for performing certain special operations.

The fourth new class is a buffer class, `strstreambuf`, which is derived from `streambuf`. The buffers used by the streams mentioned above are of class `strstreambuf`, and most of the facilities provided by the stream classes make use of functions which are members of `strstreambuf`. Most users will not need to use `strstreambuf` members, but will use the stream classes instead; such users can disregard most of the discussion of `strstreambuf` in this section.

A `strstreambuf` is a `streambuf` that uses an array of bytes (a string) to hold the sequence of characters. Given the convention that a `char*` should be interpreted as pointing just before the `char` it really points at, the mapping between the abstract get/put pointers and `char*` pointers is direct. Moving the pointers corresponds exactly to incrementing and decrementing the `char*` values. See section 6.7 for further discussion of this.

To accommodate the need for arbitrary length strings `strstreambuf` supports a dynamic mode. When a `strstreambuf` is in dynamic mode, space for the character sequence is allocated as needed. When the sequence is extended too far, it will be copied to a new array.

6.5.1 Constructors

6.5.1.1 Stream Constructors

`istrstream`	`istrstream` constructor

```
istrstream(char *cp);
```

Characters will be fetched from the (null-terminated) string **cp**. The terminating null character will not be part of the sequence. Positioning the get pointer using **istrstream::seekg** is allowed within that space.

`istrstream`	`istrstream` constructor

```
istrstream(char *cp, int len);
```

Characters will be fetched from the array beginning at **cp** and extending for **len** bytes. Positioning the get pointer using **istrstream::seekg** are allowed within that space.

`ostrstream`	`ostrstream` constructor

```
ostrstream();
```

Space will be dynamically allocated to hold stored characters.

`ostrstream`	`ostrstream` constructor

```
ostrstream(char *cp, int len, int mode);
```

Characters will be stored into the array starting at **cp** and continuing for **len** bytes.

The value of the `mode` parameter is described in the discussion of `filebuf::open` in section 6.4.2. If `ios::ate` or `ios::app` is set in `mode`, cp is assumed to be a null-terminated string and storing will begin at the null character. Otherwise, storing will begin at cp. The put pointer may be positioned to any location within the array, using `ostream::seekp`.

strstream	strstream constructor

```
strstream();
```

Space will be dynamically allocated to hold stored characters.

strstream	strstream constructor

```
strstream(char *cp, int len, int mode);
```

Characters will be stored into the array starting at cp and continuing for `len` bytes.

The value of the `mode` parameter is described in the discussion of `filebuf::open` in section 6.4.2. If `ios::ate` or `ios::app` is set in `mode`, cp is assumed to be a null-terminated string and storing will begin at the null character. Otherwise, storing will begin at cp. The put and get pointers may be positioned to any location within the array, using `istream::seekg` and `ostream::seekp`.

6.5.1.2 Buffer Constructors

strstreambuf	strstreambuf constructor

```
strstreambuf();
```

Constructs an empty strstreambuf in dynamic mode. This means that space will be automatically allocated to accommodate the characters that are put into the strstreambuf (using operators new and delete). Because this may require copying the original characters, it is recommended that when many characters will be inserted, the program should use setbuf (described below) to inform the strstreambuf.

strstreambuf	strstreambuf constructor

```
strstreambuf(void *(*a)(long),
        void (*f)(void*));
```

Constructs an empty strstreambuf in dynamic mode. In this case, the user supplies a function a to be used as the allocator function in dynamic mode. The argument passed to a will be a long denoting the number of bytes to be allocated. If a is NULL, operator new will be used. The user also supplies a function f is used to free (or delete) areas returned by a. The argument to f will be a pointer to the array allocated by a. If f is NULL, operator delete is used.

strstreambuf	strstreambuf constructor

```
strstreambuf(int n);
```

Constructs an empty strstreambuf in dynamic mode. The initial allocation of space will be at least n bytes.

strstreambuf	strstreambuf constructor

```
strstreambuf(char *b, int n, char *pstart);
strstreambuf(unsigned char *b, int n,
      unsigned char *pstart);
```

Constructs a `strstreambuf` to use the bytes starting at `b`. The `strstreambuf` will be in static mode; it will not grow dynamically. If `n` is positive, then the `n` bytes starting at `b` are used as the `strstreambuf`. If `n` is zero, `b` is assumed to point to the beginning of a null terminated string and the bytes of that string (not including the terminating null character) will constitute the `strstreambuf`. If `n` is negative, the `strstreambuf` is assumed to continue indefinitely. The get pointer is initialized to `b`. The put pointer is initialized to `pstart`. If `pstart` is `NULL`, then stores will be treated as errors. If `pstart` is not `NULL`, then the initial sequence for fetching (the get area) consists of the bytes between `b` and `pstart`. If `pstart` is `NULL`, then the initial get area consists of the entire array.

6.5.2 Stream Operations

rdbuf	return address of strstreambuf

```
strstreambuf *rdbuf();
```

Each of the three stream classes has a member `rdbuf`. This function returns the address of the `strstreambuf` associated with the stream.

str	freeze the array

```
char *str();
```

The classes `ostrstream` and `strstream` each have a member `str`.

The function returns a pointer to the array being used and "freezes" the array. Once `str` has been called the effect of storing more characters into the stream is undefined. If the stream was constructed with an explicit array, the function returns a pointer to the array. Otherwise, the address of a dynamically allocated area is returned. Until `str` is called, deleting the dynamically allocated area is the responsibility of the stream. After `str` returns, the array becomes the responsibility of the user program.

`pcount`	number of bytes stored

```
int pcount();
```

This function is a member of the `ostrstream` class. It returns the number of bytes that have been stored into the buffer. This is mainly of use when binary data has been stored and the stream's `str` member does not point to a null terminated string.

6.5.3 Buffer Operations

These functions are members of the `strstreambuf` class. As was noted above, most users will not need to use these functions, but will use the stream facilities instead.

`freeze`	freeze the buffer

```
void freeze(int n);
```

Inhibits (when `n` is nonzero) or permits (when `n` is zero) automatic deletion of the current array. When deletion is inhibited, the `strstreambuf` is said to be "frozen".

Deletion normally occurs when more space is needed or when the `strstreambuf` is being destroyed. Only space obtained via dynamic allocation is ever freed. It is an error (and the effect is undefined) to store characters into a **strstreambuf** that was in dynamic allocation mode and is now frozen. It is possible, however, to thaw (unfreeze) such a **strstreambuf** and resume storing characters.

str	freeze the buffer

```
char *str();
```

Returns a pointer to the first char of the current array and freezes the **strstreambuf**. If the **strstreambuf** was constructed with an explicit array, the function will return a pointer to that array. If the **strstreambuf** is in dynamic allocation mode, but nothing has yet been stored, the function may return **NULL**.

setpos	set length of dynamic allocation

```
streambuf *setbuf(char *p, int n);
```

The **strstreambuf** remembers **n** and the next time it does a dynamic mode allocation, it makes sure that at least **n** bytes are allocated. The **p** parameter should be **0**.

6.6 Operations on FILE Structures

This section describes facilities provided to enable stream operations to be carried out on C **FILE** structures, as declared in the C **stdio.h** header and implemented by the C run-time library.

These facilities are intended to be used when mixing C and C++ code. New C++ code should avoid using them, as the facilities described in section 6.4 give better performance.

This section describes the class `stdiobuf`, which is derived from `streambuf`. Users wishing to use these facilities should construct an `iostream` specifying a `stdiobuf` object as the `streambuf` to use.

Operations on a `stdiobuf` are reflected on the associated FILE. A `stdiobuf` is constructed in unbuffered mode, which causes all operations to be reflected immediately in the FILE. Calls to `seekg` and `seekp` are translated into call on the C run-time library function `fseek`. If required, `setbuf` can be used to supply a reserve area, which will cause buffering to be turned back on.

6.6.1 Constructor

`stdiobuf`	constructor for `stdiobuf`

```
stdiobuf(FILE *f);
```

Constructs a `stdiobuf` and connects it to the `stdio.h` FILE structure specified in `f`.

6.6.2 Other Members

`stdiofile`	pointer to FILE

```
FILE *stdiofile();
```

This function returns a pointer to the associated FILE structure.

6.7 Interfaces to streambuf

This section describes the ways in which programmers can make use of the facilities of `streambuf`, either directly, or when building derived classes of their own. Most users will not need to study this information in detail.

The **streambuf** class supports buffers into which characters can be *inserted* (or *stored*) or from which characters can be *extracted* (or *fetched*). Abstractly, such a buffer is a sequence of characters together with one or two pointers (a *get pointer* and/or a *put pointer*) that define the locations at which characters are to be stored or fetched. The pointers should be thought of as pointing between characters rather than at them. This makes it easier to understand the boundary conditions (a pointer before the first character or after the last). Some of the effects of getting and putting are defined by this class but most of the details are left to specialized classes derived from **streambuf**. For details of such derived classes, see sections 6.4, 6.5 and 6.6.

Classes derived from **streambuf** vary in their treatments of the get and put pointers. The simplest are unidirectional buffers which permit only gets or only puts. Such classes serve as pure sources (producers) or sinks (consumers) of characters. Queue-like buffers such as **strstream** (see section 6.5) have a put and a get pointer which move independently of each other. In such buffers characters that are stored are held (i.e., queued) until they are later fetched. File-like buffers such as **filebuf** (see section 6.4) permit both gets and puts but have only a single pointer. (An alternative description is that the get and put pointers are tied together so that when one moves so does the other.)

The rest of this section is devoted to three topics.

1. The **streambuf** constructors.

2. Function members intended for users who are accessing **streambuf** objects directly. This is referred to as the *public interface* to **streambuf**.

3. Function members intended for users who are building derived classes. This referred to as the *protected interface*.

Notice that some members are described both in the public and the protected interfaces.

6.7.1 Constructors

As the copying of `streambuf` objects is not regarded as well-defined, the class contains private declarations of a constructor with a `streambuf` parameter and an assignment operator. As these are private, any reference to them will be flagged as an error by the compiler.

The constructors should logically be protected. For compatibility with the old stream package, however, they are declared public.

`streambuf`	constructor for `streambuf`

```
streambuf();
```

Constructs an empty buffer, corresponding to an empty sequence of characters.

`streambuf`	constructor for `streambuf`

```
streambuf(char* ptr, int len);
```

Constructs an empty buffer and then sets up the reserve area to be the `len` bytes starting at `ptr`.

6.7.2 The Public Interface

Most `streambuf` member functions are organized into two phases.

1. As far as possible, operations are performed inline by storing into or fetching from two arrays, the *get area* and the *put area*, which together form a buffer called the *reserve area*.

2. When necessary, virtual functions are called to cope with the get and put areas. In the case of the put area, characters stored there must be flushed out into a *sink*. Conversely, characters must be read from a *source* in order to fill up the get area. Sinks and sources may be, for example, files, MS-DOS standard streams or areas of memory.

Generallythe user of a `streambuf` does not have to know anything about these details, but some of the public members pass back information about the state of the areas.

in_avail	characters available for fetching

```
int in_avail();
```

Returns the number of characters that are immediately available in the get area for fetching. This number of characters may be fetched with a guarantee that no errors will be reported.

out_waiting	characters waiting for output

```
int out_waiting();
```

Returns the number of characters in the put area that have not been output to the sink.

sbumpc	get character

```
int sbumpc();
```

Moves the get pointer forward one character and returns the character it moved past. Returns EOF if the get pointer is currently at the end of the sequence.

seekoff	position by offset

```
streampos seekoff(streamoff off, ios::seek_dir dir,
        int mode);
```

Repositions the get and/or put pointers.

The mode specifies whether the put pointer (ios::out bit set) or the get pointer (ios::in bit set) is to be modified. Both bits may be set in which case both pointers should be affected. These bits are specified with enum values defined within the ios class.

The position to move to is calculated by applying the signed byte offset parameter off to the base location specified in dir. Descriptions of the possible values for dir, and of the streampos and streamoff types, can be found in section 6.2.2.2.

Not all classes derived from streambuf support repositioning. The seekoff function will return EOF if the class does not support repositioning. If the class does support repositioning, seekoff will return the new position or EOF on error.

seekpos	position

```
streampos seekpos(streampos pos, int mode);
```

Repositions the streambuf get and/or put pointer to pos. mode specifies which pointers are affected, as for seekoff. Returns pos (the argument) or EOF if the class does not support repositioning or an error occurs.

sgetc	get character

```
int sgetc();
```

Returns the character after the get pointer. Note that it does *not* move the get pointer. Returns `EOF` if there is no character available.

setbuf	set up reserve area

```
streambuf* setbuf(char* ptr, int len);
streambuf* setbuf(unsigned char* ptr, int len);
```

Offers the `len` bytes starting at `ptr` as the reserve area. If `ptr` is `NULL` or `len` is zero or less, then an unbuffered state is requested. Whether the offered area is used, or a request for unbuffered state is honoured depends on details of the derived class. The function normally returns a pointer to the `streambuf`, but if it does not accept the offer or honour the request, it returns `0`.

sgetn	get characters

```
int sgetn(char* ptr, int n);
```

Fetches the `n` characters following the get pointer and copies them to the area starting at `ptr`. When there are fewer than `n` characters left before the end of the sequence `sgetn` fetches whatever characters remain. The function repositions the get pointer following the fetched characters and returns the number of characters fetched.

`snextc`	next character

```
int snextc();
```

Moves the get pointer forward one character and returns the character following the new position. It returns `EOF` if the pointer is currently at the end of the sequence or is at the end of the sequence after moving forward.

`sputbackc`	move get pointer back

```
int sputbackc(char c);
```

Moves the get pointer back one character. The parameter `c` must be the current contents of the character just before the get pointer. The underlying mechanism may simply back up the get pointer or may rearrange its internal data structures so that `c` is saved. Thus the effect of `sputbackc` is undefined if `c` is not the character before the get pointer. The function returns `EOF` when it fails. The conditions under which it can fail depend on the details of the derived class.

`sputc`	store character

```
int sputc(int c);
```

Stores `c` after the put pointer, and moves the put pointer past the stored character; usually this extends the sequence. It returns `EOF` when an error occurs. The conditions that can cause errors depend on the derived class.

sputn	store characters

```
int sputn(const char* ptr, int n)
```

Stores the n characters starting at ptr after the put pointer and moves the put pointer past them. The function returns the number of characters stored successfully. Normally this is the same as n, but it may be less when errors occur.

stossc	move get pointer forward

```
void stossc();
```

Moves the get pointer forward one character. If the pointer started at the end of the sequence this function has no effect.

sync	synchronise streambuf

```
int sync();
```

Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. Usually this "flushes" any characters that have been stored but not yet consumed, and "gives back" any characters that may have been produced but not yet fetched. Returns EOF to indicate errors.

6.7.3 The Protected Interface

This section describes the interface needed by programmers who are coding a derived class. Broadly speaking there are two kinds of member functions described here. The non-virtual functions are provided for manipulating a `streambuf` in ways that are appropriate in a derived class. Their descriptions reveal details of the implementation that would be inappropriate in the public interface. The virtual functions permit the derived class to specialize the `streambuf` class in ways appropriate to the specific sources and sinks that it is implementing. The descriptions of the virtual functions explain the obligations of the virtuals of the derived class. If the virtuals behave as specified, the `streambuf` will behave as specified in the public interface. However, if the virtuals do not behave as specified, then the `streambuf` may not behave properly, and a stream object (or any other code) that relies on proper behaviour of the `streambuf` may not behave properly either.

6.7.3.1 The Get, Put, and Reserve Area

The protected members of `streambuf` present an interface to derived classes organized around three areas (arrays of bytes) managed cooperatively by the base and derived classes. They are the get area, the put area, and the reserve area (or buffer). The get and the put areas are normally disjoint, but they may both overlap the reserve area, whose primary purpose is to be a resource in which space for the put and get areas can be allocated. The get and the put areas are changed as characters are fetched from and stored in the buffer, but the reserve area normally remains fixed. The areas are defined by a collection of `char*` values. The buffer abstraction is described in terms of pointers that point between characters, but the `char*` values must point at chars. To establish a correspondence the `char*` values should be thought of as pointing just before the byte they really point at.

Functions to Examine the Pointers

`base`	start of reserved area

```
char* base();
```

Returns a pointer to the first byte of the reserve area. Space between **base** and **ebuf** is the reserve area.

eback	limit of putback

```
char* eback();
```

Returns a pointer to a lowest allowable location for **gptr**. Space between **eback** and **gptr** is available for putback.

ebuf	end of reserve area

```
char* ebuf();
```

Returns a pointer to the byte after the last byte of the reserve area.

egptr	end of get area

```
char* egptr();
```

Returns a pointer to the byte after the last byte of the get area.

epptr	end of put area

```
char* epptr();
```

Returns a pointer to the byte after the last byte of the put area.

gptr	start of get area

```
char* gptr();
```

Returns a pointer to the first byte of the get area. The available characters are those between `gptr` and `egptr`. The next character fetched will be `*gptr` unless `egptr` is less than or equal to `gptr`.

pbase	base of put area

```
char* pbase();
```

Returns a pointer to the put area base. Characters between `pbase` and `pptr` have been stored into the buffer and not yet consumed.

pptr	start of put area

```
char* pptr();
```

Returns a pointer to the first byte of the put area. The space between `pptr` and `epptr` is the put area and characters will be stored here.

Functions for Setting the Pointers

Note that to indicate that a particular area (get, put, or reserve) does not exist, all the associated pointers should be set to zero.

setb	define reserve area

```
void setb(char* b, char* eb, int i);
```

Sets `base` and `ebuf` to `b` and `eb` respectively. The `i` parameter controls whether the area will be subject to automatic deletion. If `i` is non-zero, then `b` will be deleted when `base` is changed by another call on `setb`, or when the destructor is called for the `streambuf`. If `b` and `eb` are both `NULL` then we say that there is no reserve area. If `b` is not `NULL`, there is a reserve area even if `eb` is less than `b` and so the reserve area has zero length.

setp	define put area

```
void setp(char* p, char* ep);
```

Sets `pptr` and `pbase` to `p`, and `epptr` to `ep`.

setg	define get area

```
void setg(char* eb, char* g, char* eg);
```

Sets `eback` to `eb`, `gptr` to `g`, and `egptr` to `eg`.

6.7.3.2 Other Non-Virtual Members

allocate	set up reserve area

```
int allocate();
```

Tries to set up a reserve area. If a reserve area already exists or if `unbuffered` is non-zero, `allocate` returns 0 without doing anything. If the attempt to allocate space fails, `allocate` returns `EOF`; otherwise it returns 1. `allocate` is not called by any non-virtual member function of `streambuf`.

blen	size of reserve area

```
int blen();
```

Returns the size of the current reserve area.

dbp	print debug information

```
void dbp();
```

Writes directly on to `stdout` information in ASCII about the state of the buffer. It is intended for debugging and nothing is specified about the form of the output. It is considered part of the protected interface because information it prints can only be understood in relation to that interface, but it is a public function so that it can be called anywhere during debugging.

gbump	increment gptr

```
void gbump(int n);
```

Increments `gptr` by `n`, which may be positive or negative. No checks are made on whether the new value of `gptr` is in bounds.

pbump	increment **pptr**

```
void pbump(int n);
```

Increments **pptr** by **n**, which may be positive or negative. No checks are made on whether the new value of **pptr** is in bounds.

unbuffered	buffering state

```
int unbuffered();
void unbuffered(int i);
```

A **streambuf** includes a private variable which holds the **streambuf**'s *buffering state*. The call **unbuffered(i)** sets the value of this variable to **i**. The call **unbuffered()** returns the current value. This state is independent of the actual allocation of a reserve area. Its primary purpose is to control whether a reserve area is allocated automatically by **allocate**.

6.7.3.3 Virtual Member Functions

Virtual functions may be redefined in derived classes to specialize the behaviour of **streambufs**. This section describes the behaviour that these virtual functions should have in any derived classes; the next section describes the behaviour that these functions are defined to have in base class **streambuf**.

doallocate	perform allocation

```
int doallocate();
```

This function is called when **allocate** determines that space is needed. It is required to call **setb** to provide a reserve area or to return **EOF** if it cannot. It is only called if **unbuffered** is zero and **base** is zero.

overflow	consume characters

```
int overflow(int c);
```

This function is called to consume characters, that is, to send them to their ultimate sink, for example, a file. If c is not **EOF**, overflow also must either save c or consume it. Usually it is called when the put area is full and an attempt is being made to store a new character, but it can be called at other times. The normal action is to consume the characters between **pbase** and **pptr**, call **setp** to establish a new put area, and if c is not **EOF** store it (using **sputc**). This function should return **EOF** to indicate an error; otherwise it should return something else.

pbackfail	handle putback failure

```
int pbackfail(int c);
```

This is called when **eback** equals **gptr** and an attempt has been made to putback c. If this situation can be dealt with (e.g., by repositioning an external file), **pbackfail** should return c; otherwise it should return **EOF**.

seekoff	position by offset

```
streampos seekoff(streamoff off, ios::seek_dir dir,
        int mode);
```

Repositions the get and/or put pointers (i.e., the abstract get and put pointers, not `pptr` and `gptr`). The meanings of `off` and `dir` are discussed in section 6.7.2. The `mode` parameter specifies whether the put pointer (`ios::out` bit set) or the get pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. A class derived from `streambuf` is not required to support repositioning.

The function should return `EOF` if the class does not support repositioning. If the class does support repositioning, `seekoff` should return the new position or `EOF` on error.

`seekpos`	position

```
streampos seekpos(streampos pos, int mode);
```

Repositions the `streambuf` get and/or put pointer to `pos`. The `mode` parameter specifies which pointers are affected, as for `seekoff`. Returns `EOF` if the class does not support repositioning or an error occurs; otherwise, returns the value of the `pos` parameter.

`setbuf`	establish reserve area

```
streambuf* setbuf(char* ptr, int len);
streambuf* setbuf(unsigned char* ptr, int len);
```

Offers the array at `ptr` with `len` bytes to be used as a reserve area. The normal interpretation is that if `ptr` or `len` are zero then this is a request to make the `streambuf` unbuffered. The derived class may use this area or not as it chooses. It may accept or ignore the request for unbuffered state as it chooses. The function should return a pointer to the `streambuf` if it honours the request; otherwise it should return 0.

`sync`	synchronise `streambuf`

```
int sync();
```

This function is called to give the derived class a chance to look at the state of the areas, and synchronise them with any external representation. Normally `sync` should consume any characters that have been stored into the put area, and if possible give back to the source any characters in the get area that have not been fetched. When `sync` returns there should not be any unconsumed characters, and the get area should be empty. It should return `EOF` if some kind of failure occurs.

`underflow`	supply characters

```
int underflow();
```

This is called to supply characters for fetching, i.e., to create a condition in which the get area is not empty. These characters would be obtained from the ultimate source; for example, a file. If it is called when there are characters in the get area it should return the first character. If the get area is empty, it should create a nonempty get area and return the next character (which it should also leave in the get area). If there are no more characters available, `underflow` should return `EOF` and leave an empty get area.

6.7.3.4 Default Definitions of the Virtual Functions

This section describes the behaviour of the versions of the virtual functions which are actually members of `streambuf`. These are the ones used, for example, by the core stream classes, `istream`, `ostream` and `iostream`.

`streambuf::doallocate`	perform allocation

```
int doallocate();
```

Attempts to allocate a reserve area using the operator `new`.

`streambuf::overflow`	consume characters

```
int overflow(int c);
```

Its behaviour is compatible with the old stream package, but that behaviour is not considered part of the specification of the current stream package. Therefore, `streambuf::overflow` should be treated as if it had undefined behaviour, and should always be defined in derived classes.

`streambuf::pbackfail`	handle putback failure

```
int pbackfail(int c);
```

Always returns `EOF`.

`streambuf::seekpos`	position

```
streampos seekpos(streampos pos, int mode);
```

Returns `seekoff(streamoff(pos),ios::beg, mode)`. Thus to define seeking in a derived class, it is frequently only necessary to define `seekoff` and use the inherited `streambuf::seekpos`.

`streambuf::seekoff`	position by offset

```
streampos seekoff(streamoff off,ios::seek_dir dir,
     int mode);
```

Always returns EOF; in other words, `streambuf` itself does not support positioning.

`streambuf::setbuf`	establish reserved area

```
streambuf* setbuf(char* ptr, int len);
```

Honours the request when there is no reserve area.

`streambuf::sync`	synchronise `streambuf`

```
int sync();
```

Returns 0 if the get area is empty and there are no unconsumed characters. Otherwise it returns EOF.

`streambuf::underflow`	consume characters

```
int underflow();
```

Its behaviour is compatible with the old stream package, but that behaviour is not considered part of the specification of the current stream package. Therefore, `streambuf::underflow` should be treated as if it had undefined behaviour, and should always be defined in derived classes.

Appendix A

Distribution Kit

This appendix lists the files which are installed on the user's hard disk when the process described in chapter 1 is followed. Each file name is accompanied by a short description of the file's function.

Note that these files are those added to an existing Parallel C kit. The files which are part of the Parallel C product are not listed.

A.1 Directory \tc2v2

`cpp.b4`	C++ preprocessor
`cfront.b4`	C++ front-end processor
`t4cc.exe`	compiler driver for T4 mode
`t8cc.exe`	compiler driver for T8 mode
`libct4.bin`	`iostream` class library for T4
`libct8.bin`	`iostream` class library for T8
`complxt4.bin`	`complex` class library for T4
`complxt8.bin`	`complex` class library for T8
`t4cclink.bat`	linker batch file for T4
`t8cclink.bat`	linker batch file for T8
`t4cctask.bat`	batch file to link task for T4

`t8cctask.bat`	batch file to link task for T8
`t4ccstas.bat`	batch file to link a stand-alone task for T4
`t8ccstas.bat`	batch file to link a stand-alone task for T8

A.2 Directory \tc2v2\cc

`alt.h`	`ascii.h`	`assert.h`
`boot.h`	`chan.h`	`chanio.h`
`common.h`	`ctype.h`	`dos.h`
`errno.h`	`float.h`	`fstream`
`generic`	`iomanip`	`iostream`
`limits.h`	`locale.h`	`malloc.h`
`math.h`	`memory.h`	`net.h`
`new.h`	`osfcn.h`	`ostream.h`
`par.h`	`sema.h`	`serv.h`
`setjmp.h`	`signal.h`	`stdarg.h`
`stddef.h`	`stdio.h`	`stdiostr.h`
`stdlib.h`	`stream.h`	`string.h`
`strstea.h`	`thread.h`	`time.h`
`timer.h`	`varargs.h`	`vector.h`
`values.h`		

A.3 Directory \tc2v2\examples

`hello.cpp`	"Hello, world." program

Appendix B

Summary of Option Switches

This appendix lists the C++ option switches. Further information can be found in section 4.2, in the subsections specified below for each switch. For similar listings for the linker and server, see appendix D of the Parallel C *User Guide*[6].

In the table below, the following notations are used to describe the formats of the switches.

fn	An MS-DOS filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 4.2.
dir	An MS-DOS filename, which will be assumed to refer to a directory.
mac	Any sequence of characters which is acceptable to the compiler as a macro name.
str	Any sequence of characters which is acceptable to the compiler as the value of a macro.

n	A decimal integer.

Switches and their arguments are not case sensitive, except as noted in section 4.2.

`/C`	4.2.2 Check: do not generate object file.
`/D`*mac*	4.2.6 Define *mac* with the value `1`.
`/D`*mac*=*str*	4.2.6 Define *mac* with the value *str*.
`/FB`*fn*	4.2.1 Put binary object output in *fn*.
`/FO`*fn*	4.2.1 Identical to `/FB`.
`/GD`	4.2.2 Perform all floating-point arithmetic in double precision.
`/I`	4.2.7 Print the compiler's identification.
`/I`*dir*	4.2.5 Add *dir* to the `#include` list.
`/PC`*n*	4.2.3 Set the number of bytes required for an `extern` function call.
`/PM`*n*	4.2.3 Set the number of bytes required for a module number.
`/S`	4.2.2 Perform floating arithmetic in single precision (ignored).
`/U`*mac*	4.2.6 Undefine a predefined macro.
`/W`	4.2.7 Suppress most warnings.
`/ZD`	4.2.4 Generate line number tables for `decode` and debugger.
`/ZI`	4.2.4 Generate line number tables and debug tables for variables.
`/ZO`	4.2.4 Generate old format diagnostic information.

Bibliography

[1] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986. ISBN 0-201-12078-X.

[2] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[3] Al Stevens. *Teach Yourself C++*. MIS Press, 1990. ISBN 1-55828-027-8.

[4] Stephen C. Dewhurst and Kathy T. Stark. *Programming in C++*. Prentice-Hall, 1989. ISBN 0-13-723156-3.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, First Edition*. Prentice-Hall, 1978. ISBN 0-13-110163-3.

[6] *Parallel C User Guide*. 3L Ltd, 1991. Software version 2.2.2.

[7] *Disk Operating System Version 3.10 Reference*. International Business Machines, February 1985.

[8] *Microsoft MS-DOS User's Reference*. Microsoft Corporation, 1986. Document Number 410630013-320-R03-0686.

[9] *Disk Operating System Version 3.00 Technical Reference*. International Business Machines, May 1984.

[10] *Stand alone compiler implementation manual*. Version 1.1, Inmos Ltd., July 1987.

[11] *TDS Compiler implementation manual.* Version 1.0, Inmos Ltd., November 19, 1986.

Index

D

E

F

T

U

V

W

X