UNIT - IV
C++ I/O

C++ supports two complete I/O systems.

- Inherits from C.
- Object-oriented I/O system defined by C++

NOTE: C++ programs can also use the C++-style header #include<cstdio>
.

I/O using C functions

It includes,

- Console I/O
- Streams
- Files

Console I/O

This can be divided into Unformatted and Formatted Console I/O.

Unformatted Console I/O

a. **Reading and Writing Characters**

The simplest of the console I/O functions are getchar( ) and putchar().

getchar( ) : It reads a character from the keyboard. It waits until a key is pressed and then returns its value. The key pressed is also automatically echoed to the screen.

Prototype: int getchar(void);

putchar( ): It prints or writes a character to the screen at the current cursor position.

Prototype: int putchar(int c);

Program:

```
#include <iostream>
#include<cstdio>
using namespace std;

int main()
{
    char ch;
    cout<<"\n Enter a character in lower case: ";
    ch = getchar();
    cout<<"\nThe entered character is ";

    putchar(ch);
    cout<<"\nCharacter in UPPER CASE: ";
    putchar(ch - 32);
    return 0;
}
```

Output:

```
    Enter a character in lower case: t
    The entered character is t
    Character in UPPER CASE: T
```

b. **Alternatives to getchar( )**

getchar( ) is not useful in an interactive environment. Two of the most common alternative functions, getch( ) and getche( )

getch( ) : It waits for a keypress, after which it returns immediately. It does not echo the character to the screen.
Prototype: int getch(void);

getche( ) : It is the same as getch( ), but the key is echoed.
Prototype: int getche(void);

c. Reading and Writing Strings

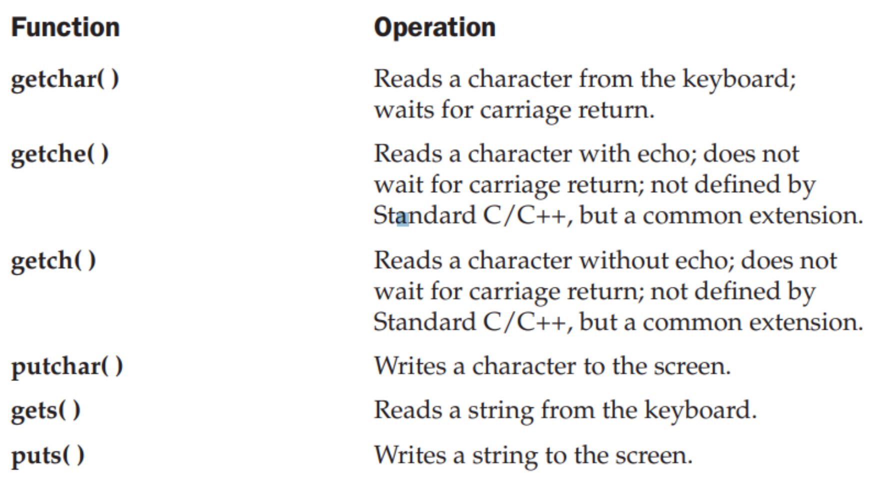
gets( ) : It reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. You may type characters at the keyboard until you press ENTER. The carriage return does not become part of the string; instead, a null terminator is placed at the end and gets( ) returns.
Prototype: char *gets(char *str);

Problem with gets( ) : It performs no boundary checks on the array that is receiving input. Thus, it is possible for the user to enter more characters than the array can hold. One alternative is the fgets( ) function.

puts( ): It writes its string argument to the screen followed by a newline.
Prototype: int puts(const char *str);

Program:
```
include <iostream>
#include<cstdio>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    gets(str);
    cout << "You entered: " << str;

    char str1[] = "Happy New Year";
    char str2[] = "Happy Birthday";

    puts(str1);
    /*  Printed on new line since '/n' is added */
    puts(str2);

    return 0;
}
```

Output:
```
main.cpp:18:5: warning: ‘char* gets(char*)’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.cpp:18:13: warning: ‘char* gets(char*)’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.cpp:(.text+0x31): warning: the `gets' function is dangerous and should not be used.
Enter a string: rt
You entered: rtHappy New Year
Happy Birthday
```



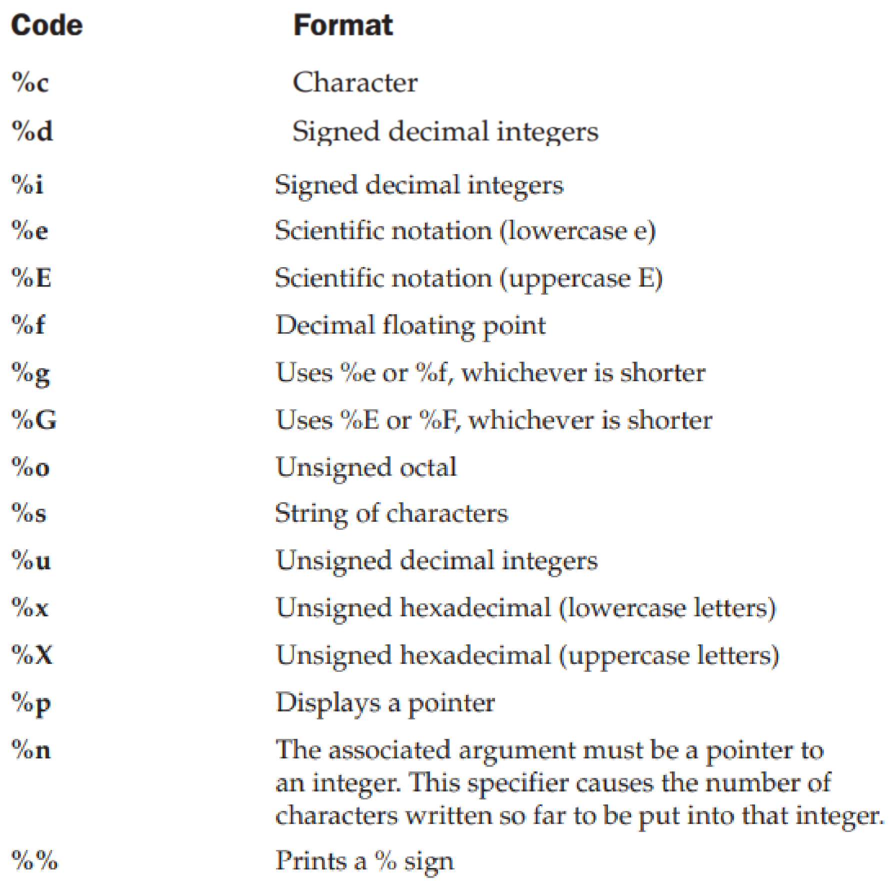
Function	Operation
getchar( )	Reads a character from the keyboard; waits for carriage return.
getche( )	Reads a character with echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
getch( )	Reads a character without echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
putchar( )	Writes a character to the screen.
gets( )	Reads a string from the keyboard.
puts( )	Writes a string to the screen.

Formatted Console I/O

The functions printf( ) and scanf( ) perform formatted output and input. Both functions can operate on any of the built-in data types, including characters, strings, and numbers.

printf( ): It writes data to the console.
Prototype: int printf(const char *control_string, ...);

The control_string consists of two types of items. The first type is composed of characters that will be printed on the screen. The second type contains format specifiers that define the way the subsequent arguments are displayed. A format specifier begins with a percent sign and is followed by the format code.



Code	Format
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%%	Prints a % sign

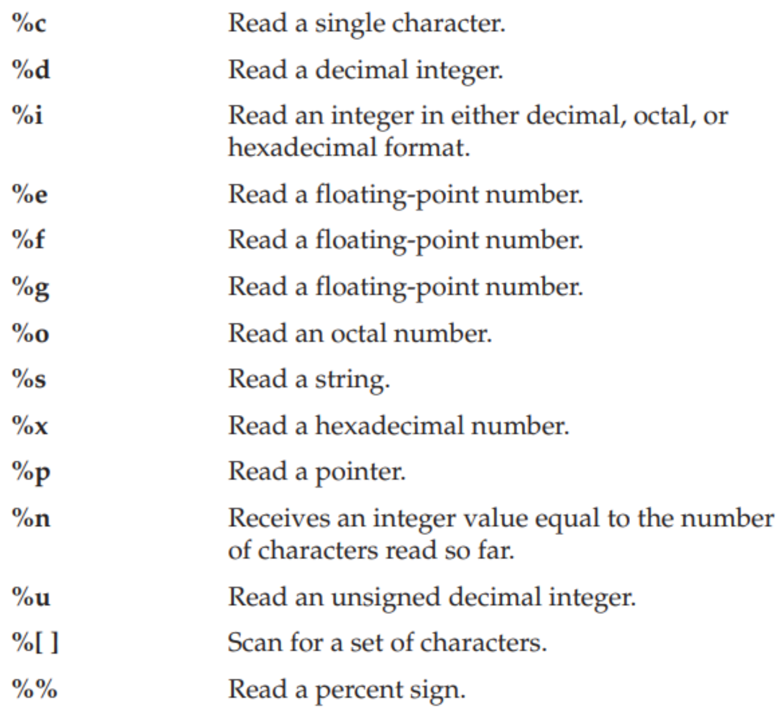
scanf( ): Its reads data from the keyboard.
Prototype: int scanf(const char *control_string, ...);

The control_string determines how values are read into the variables pointed to in the argument list. The control string consists of three classifications of characters:

- Format specifiers
- White-space characters
- Non-white-space characters

Format specifiers

The input format specifiers are preceded by a % sign and tell scanf( ) what type of data is to be read next.



%c	Read a single character.
%d	Read a decimal integer.
%i	Read an integer in either decimal, octal, or hexadecimal format.
%e	Read a floating-point number.
%f	Read a floating-point number.
%g	Read a floating-point number.
%o	Read an octal number.
%s	Read a string.
%x	Read a hexadecimal number.
%p	Read a pointer.
%n	Receives an integer value equal to the number of characters read so far.
%u	Read an unsigned decimal integer.
%[ ]	Scan for a set of characters.
%%	Read a percent sign.

White-space characters

A white-space character in the control string causes scanf( ) to skip over one or more leading white-space characters in the input stream. A white-space character is a space, a tab, vertical tab, form feed, or a newline.

Non-white-space characters

A non-white-space character in the control string causes scanf( ) to read and discard matching characters in the input stream. For example, "%d,%d" causes scanf( ) to read an integer, read and discard a comma, and then read another integer

Program:

```
#include<iostream>
#include<cstdio>
using namespace std;

int main()
{
    int f;
    printf("  ff");
    scanf("%d",f);
    printf(f);
    return 0;
}
```

Output: ff f

Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. The file system transforms each into a logical device called a stream. There are two types of streams: text and binary.

- **Text Streams**

A text stream is a sequence of characters. Standard C allows (but does not require) a text stream to be organized into lines terminated by a newline character.

Certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

- **Binary Streams**

A binary stream is a sequence of bytes that have a one-to-one correspondence to those in the external device that is, no character translations occur. Also, the number of bytes written (or read) is the same as the number on the external device.

The Standard Streams

As it relates to the C file system, when a program starts execution, three streams are opened automatically. They are stdin (standard input), stdout (standard output), and stderr (standard error).

Using freopen( ) to Redirect the Standard Streams

You can redirect the standard streams by using the freopen( ) function. This function associates an existing stream with a new file. Thus, you can use it to associate a standard stream with a new file.

Prototype: FILE *freopen(const char *filename, const char *mode, FILE *stream);

filename is a pointer to the filename you wish associated with the stream pointed to by stream. The file is opened using the value of mode, which may have the same values as those used with fopen( ). freopen( ) returns stream if successful or NULL on failure.

Program

```
#include <cstdio>
#include <cstdlib>

int main()
{
    FILE* fp = fopen("test1.txt","w");
    fprintf(fp,"%s","This is written to test1.txt");

    if (freopen("test2.txt","w",fp))
    fprintf(fp,"%s","This is written to test2.txt");
    else
    {
    printf("freopen failed");
    exit(1);
}

    fclose(fp);
    return 0;
}
```

Output:

Files

In C/C++, a *file* may be anything from a disk file to a terminal or printer. Each stream that is associated with a file has a file control structure of type **FILE**.

- **File System Basics**

The C file system is composed of several interrelated functions. C++ programs may also use the C++-style header **<cstdio>**.

- **The File Pointer**

The file pointer is the common thread that unites the C I/O system. A *file pointer* is a pointer to a structure of type **FILE**. It points to information that defines various things about the file, including its name, status, and the current position of the file.

In order to read or write files, your program needs to use file pointers

Prototype : FILE *fp;

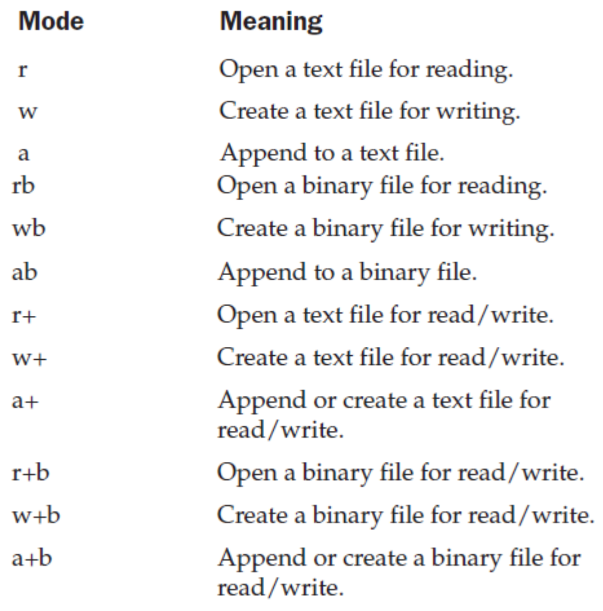
File Operations

- **fopen( )** : It opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file.

Prototype: FILE *fopen(const char **filename*, const char **mode*);

where *filename* is a pointer to a string of characters that make up a valid filename and may include a path specification.

The legal values for *mode are,*



Mode	Meaning
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

- **fclose( ):** It closes a stream that was opened by a call to **fopen( )**.

Prototype: int fclose(FILE **fp*);

where *fp* is the file pointer returned by the call to **fopen( )**. The function returns **EOF** if an error occurs.

Program:

```
#include <cstdio>
#include <cstring>
```

```
#include<iostream>
using namespace std;

int main()
{
    int c;
    FILE *fp;
    fp = fopen("file.txt", "w+r");
    char str[20] = "Hello World!";
    if (fp)
    {
        for(int i=0; i<strlen(str); i++)
        putc(str[i],fp);
    }
    fclose(fp);
}
```

Output:

```
        Hello World!
```

- **putc( )** and **fputc( )**

These two equivalent functions writes characters to a file that was previously opened for writing using the **fopen( )** function.

Prototype : int putc(int *ch*, FILE **fp*);

where *fp* is the file pointer returned by **fopen( )** and *ch* is the character to be output. The file pointer tells **putc( )** which file to write to.

If a **putc( )** operation is successful, it returns the character written. Otherwise, it returns **EOF**.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    char str[] = "Testing putc() function";
    FILE *fp;

    fp = fopen("file.txt","w");

    if (fp)
    {
      for(int i=0; i<strlen(str); i++)
      {
      putc(str[i],fp);
      }

      for(int i=0; i<strlen(str); i++)
      {
      fputc(str[i],fp);
      }
    }
    else
      perror("File opening failed");
      fclose(fp);
    return 0;
}
```

Output:

```
        Testing putc() functionTesting putc() function
```

- **getc( )** and **fgetc( )**
  These two equivalent functions reads characters from a file opened in read mode by **fopen( )**.

Prototype : int getc(FILE **fp*);

where *fp* is a file pointer of type **FILE** returned by **fopen( )**. The **getc( )** function returns an **EOF** when the end of the file has been reached. **getc( )** also returns **EOF** if an error occurs.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
  int c;
  FILE *fp;

  fp = fopen("file.txt","r");

  if (fp)
  {
    while(feof(fp) == 0)
    {
    c = getc(fp);
    putchar(c);
    }
    while(feof(fp) == 0)
    {
    c = fgetc(fp);
    putchar(c);
    }
  }
  else
    perror("File opening failed");
    fclose(fp);
  return 0;
}
```

Output:

```
Testing putc() functionTesting putc() function
```

- **feof( )**: It determines when the end of the file has been encountered.

Prototype: int feof(FILE **fp*);

feof( ) returns true if the end of the file has been reached; otherwise, it returns 0.

- **fputs( ) and fgets( )**

These functions work just like **putc( )** and **getc( )**, but instead of reading or writing a single character, they read or write strings.

Prototypes:

int fputs(const char **str*, FILE **fp*);
char *fgets(char **str*, int *length*, FILE **fp*);

The **fputs( )** function writes the string pointed to by *str* to the specified stream. It returns **EOF** if an error occurs.

The **fgets( )** function reads a string from the specified stream until either a newline character is read or *length* −1 characters have been read.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int count = 10;
    char str[10];
    FILE *fp;

    fp = fopen("file.txt","w+");
    fputs("An example file\n", fp);
    fputs("Filename is file.txt\n", fp);

    rewind(fp);

    while(feof(fp) == 0)
    {
        fgets(str,count,fp);
        cout << str << endl;
     }

    fclose(fp);
    return 0;
}
```

Output:

```
An exampl
e file

Filename
is file.t
xt

xt
```

- **rewind( )**

The **rewind( )** function resets the file position indicator to the beginning of the file specified as its argument. That is, it "rewinds" the file.

Prototype: void rewind(FILE **fp*);

where *fp* is a valid file pointer.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int c;
    FILE *fp;
    fp = fopen("file.txt", "r+w");
    if (fp)
    {
      while ((c = getc(fp)) != EOF)
      putchar(c);

      rewind(fp);
      putchar('\n');

      while ((c = getc(fp)) != EOF)
      putchar(c);
    }
    fclose(fp);
    return 0;
}
```

Output:

```
welcome
welcome
```

- **ferror( )**

  The **ferror( )** function determines whether a file operation has produced an error.

Prototype: int ferror(FILE **fp*);

where *fp* is a valid file pointer. It returns true if an error has occurred during the last file operation; otherwise, it returns false.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int ch;
    FILE* fp;
    fp = fopen("file.txt","w");

    if(fp)
    {
      ch = getc(fp);
      if (ferror(fp))
      cout << "Can't read from file";
    }
    fclose (fp);
    return 0;
}
```

Output:

```
Can't read from file
```

- **remove( )**

  The **remove( )** function erases the specified file.

Prototype: int remove(const char **filename*);

It returns zero if successful; otherwise, it returns a nonzero value.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    char filename[] =  "file.txt";

    /*Deletes the file if exists */
    if (remove(filename) != 0)
    perror("File deletion failed");
    else
    cout << "File deleted successfully";

    return 0;
}
```

Output:

```
File deleted successfully
```

- **fflush( )**

  If you wish to flush the contents of an output stream, use the **fflush( )** function.

Prototype: int fflush(FILE **fp*);

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int x;
    char buffer[1024];
    setvbuf(stdout, buffer, _IOFBF, 1024);
    printf("Enter an integer - ");
    fflush(stdout);
    scanf("%d",&x);
    printf("You entered %d", x);
    return(0);
}
```

Output:
```
Enter an integer - 89
You entered 89
```

- **fread( )** and **fwrite( )**.
  These functions allow the reading and writing of blocks of any type of data.

Prototypes:

size_t fread(void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);
size_t fwrite(const void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);

For **fread( )**, *buffer* is a pointer to a region of memory that will receive the data from the file. For **fwrite( )**, *buffer* is a pointer to the information that will be written to the file. The value of *count* determines how many items are read or written, with each item being *num_bytes* bytes in length. Finally, *fp* is a file pointer to a previously opened stream.

The **fread( )** function returns the number of items read. This value may be less than *count* if the end of the file is reached or an error occurs. The **fwrite( )** function returns the number of items written. This value will equal *count* unless an error occurs.

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int retVal;
    FILE *fp;
    char buffer[] = "Writing to a file using fwrite.";
    fp = fopen("data.txt","wb");

    retVal = fwrite(buffer,sizeof(buffer),1,fp);
    cout << "fwrite returned " << retVal;

    return 0;
}
```

Output:
```
fwrite returned 1
```

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
 FILE *fp;
 char buffer[100];

 fp = fopen("data.txt","rb");
 while(!feof(fp))
 {
```

```
 fread(buffer,sizeof(buffer),1,fp);
 cout << buffer;
 }

 return 0;
}
```

Output:

```
Writing to a file using fwrite.
```

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{

    FILE *fp;
    double d = 12.23;
    int i = 101;
    long l = 123023L;
    if((fp=fopen("test", "wb+"))==NULL)
    {
       printf("Cannot open file.\n");
       exit(1);
    }
    fwrite(&d, sizeof(double), 1, fp);
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&l, sizeof(long), 1, fp);
    rewind(fp);
    fread(&d, sizeof(double), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread(&l, sizeof(long), 1, fp);
    printf("%f %d %ld", d, i, l);
    fclose(fp);
    return 0;
}
```

Output:

```
12.230000 101 123023
```

- **fseek( )**

You can perform random-access read and write operations using the C I/O system with the help of **fseek( )**, which sets the file position indicator.

Prototype : int fseek(FILE **fp*, long int *numbytes*, int *origin*);

Here, *fp* is a file pointer returned by a call to **fopen( )**. *numbytes* is the number of bytes from *origin* that will become the new current position, and *origin* is one of the following macros:

Origin Macro	**Name**
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

Program:

```
#include <cstdio>
```

```
#include <cstring>
#include<iostream>
using namespace std;

int main(int argc, char *argv[])
{
      FILE * pFile;
      pFile = fopen ( "example.txt" , "wb" );
      fputs ( "This is an apple." , pFile );
      fseek ( pFile , 9 , SEEK_SET );
      fputs ( " sam" , pFile );
      fclose ( pFile );
      return 0;
}
```

Output:
After this code is successfully executed, the file example.txt contains:

```
This is a sample.
```

- **fprintf( ) and fscanf( )**
  These functions behave exactly like **printf( )** and **scanf( )** except that they operate with files.

Prototypes:

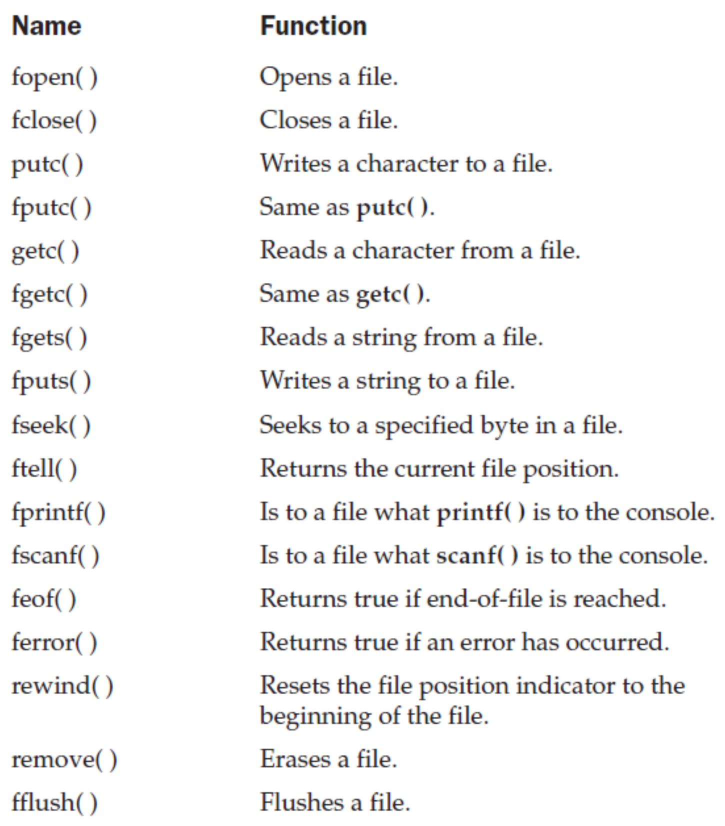
int fprintf(FILE **fp*, const char **control_string*,. . .);
int fscanf(FILE **fp*, const char **control_string*,. . .);
where *fp* is a file pointer returned by a call to **fopen( )**. **fprintf( )** and **fscanf( )** direct their I/O operations to the file pointed to by *fp*.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main(int argc, char *argv[])
{
     FILE *fp;
     char name[50];
     int age;

     fp = fopen("example.txt","w");
     fprintf(fp, "%s %d", "Tim", 9);
     fclose(fp);
     fp = fopen("example.txt","r");
     fscanf(fp, "%s %d", name, &age);
     fclose(fp);
     printf("Hello %s, You are %d years old\n", name, age);
     return 0;
}
```

Output:

```
Hello Tim, You are 9 years old
```



Name	Function
fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as **putc( )**.
getc( )	Reads a character from a file.
fgetc( )	Same as **getc( )**.
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file what **printf( )** is to the console.
fscanf( )	Is to a file what **scanf( )** is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

Object-oriented I/O system defined by C++

C++ Streams

The C++ I/O system operates through streams. A *stream* is a logical device that either produces or consumes information.

A stream is linked to a physical device by the I/O system. All streams behave the same, the same I/O functions can operate

Stream classes' hierarchy

Standard C++ provides support for its I/O system in **<iostream>** header, which gives a set of class hierarchies is defined that supports I/O operations.

The C++ I/O system is built upon two related but different class hierarchies.

- **basic_streambuf**

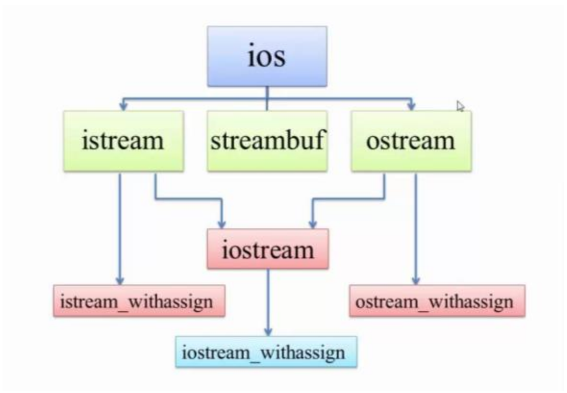
Low-level I/O class is called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly.

- **basic_ios**

The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O.

- **ios_base**

A base class for **basic_ios** is called **ios_base. basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively.



istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.
Class hierarchies for 8-bit characters and wide characters.

Class	Character-based Class	Wide-Character-based Class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

C++'s Predefined Streams

When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**. By default, the standard streams are used to communicate with the console.

Standard C++ also defines these four additional streams: **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

Operator Overloading

Overloading << and >>

<< and the >> operators are overloaded in C++ to perform I/O. the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream. The functions that overload the insertion and extraction operators are generally called *inserters* and *extractors*, respectively.

Creating Your Own Inserters

It is quite simple to create an inserter for a class that you create.

General form:

ostream &operator<<(ostream &*stream*, *class_type obj*)

```
{
    // body of inserter
    return stream;
}
```

The function returns a reference to a stream of type **ostream**. The first parameter to the function is a reference to the output stream. The second parameter is the object being inserted.

Within an inserter function, you may put any type of procedures or operations that you want. inserters cannot be members of the class for which they are defined seems to be a serious problem because they cannot access the private elements of a class. Solution is to Make the inserter a **friend** of the class

An inserter need not be limited to handling only text. An inserter can be used to output data in any form like CAD plotters, graphics images, dialog boxes etc.

Creating Your Own Extractors

Extractors are the complement of inserters.

General form

```
istream &operator>>(istream &stream, class_type &obj)
{
    // body of extractor
    return stream;
}
```

Extractors return a reference to a stream of type **istream**, which is an input stream. The first parameter must also be a reference to a stream of type **istream**. The second parameter must be a reference to an object of the class for which the extractor is overloaded. This is so the object can be modified by the input (extraction) operation.

Program:

```
#include <iostream>
#include <cstring>
using namespace std;

class Box
{
   double height;
   double width;
   double vol ;

   public :
   friend istream & operator >> (istream &, Box &);
   friend ostream & operator << (ostream &, Box &);
};

istream & operator >> (istream &stream, Box &b)
{
   cout << "Enter Box Height: " ; stream >> b.height ;
   cout << "Enter Box Width : " ; stream >> b.width ;
   return (stream) ;
}
ostream & operator << (ostream &stream, Box &b)
{
   stream << endl << endl;
   stream << "Box Height : " << b.height << endl ;
```

```
    stream << "Box Width  : " << b.width << endl ;

    b.vol = b.height * b.width ;
    stream << "The Volume of Box : " << b.vol << endl;

    return(stream) ;
}

 int main()
 {
     Box b1;
     cin >> b1;
     cout << b1;
}
```

Output:

```
Enter Box Height: 1
Enter Box Width : 2

Box Height : 1
Box Width  : 2
The Volume of Box : 2
```

Creating Your Own Manipulator Functions

We can customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons.

- You can consolidate a sequence of several separate I/O operations into one manipulator.
- When you need to perform I/O operations on a nonstandard device. For example, you might use a manipulator to send control codes to a special type of printer or to an optical recognition system.

Types of manipulators

There are two basic types of manipulators:

- Those that operate on input streams
- Those that operate on output streams.

Apart from this , there is one more classification,

- Manipulators that take an argument

  The procedures necessary to create a parameterized manipulator vary widely from compiler to compiler, and even between two different versions of the same compiler. For this reason, you must consult the documentation to your compiler for instructions on creating parameterized manipulators
- Manipulators that don't.take an argument

  The creation of parameterless manipulators is straightforward and the same for all compilers.

General Form

```
ostream &manip-name(ostream &stream)
{
    // your code here
    return stream;
}
```

manip-name is the name of the manipulator. a reference to a stream of type **ostream** is returned. This is necessary if a manipulator is used as part of a larger I/O expression.

Using an output manipulator is particularly useful for sending special codes to a device. For example, a printer may be able to accept various codes that change the type size or font, or that position the print head in a special location. If these adjustments are going to be made frequently, they are perfect candidates for a manipulator.

General Form

```
istream &manip-name(istream &stream)
{
  // your code here
  return stream;
}
```

An input manipulator receives a reference to the stream for which it was invoked. This stream must be returned by the manipulator.

Program:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>
using namespace std;

// A simple output manipulator that sets the fill character to * and sets the field width to 10.
ostream &star_fill(ostream &stream)
{
      stream << setfill('*') << setw(10);
      return stream;
}

// A simple input manipulator that skips leading digits.
istream &skip_digits(istream &stream)
{
       char ch;/*w  w  w .  j  ava 2s.c  o  m*/
       do
       {
       ch = stream.get();
       } while(!stream.eof() && isdigit(ch));

       if(!stream.eof()) stream.unget();
       return stream;
}
int main()
{
       string str;
       // Demonstrate the custom output manipulator.
       cout << 512 << endl;
       cout << star_fill << 512 << endl;
       // Demonstrate the custom input manipulator.
       cout << "Enter some characters: ";
       cin >> skip_digits >> str;
       cout << "Contents of str: " << str;
        return 0;
}
```

Output:

```
512
*******512
Enter some characters: abc
```

```
Contents of str: abc
```

File streams and String streams
String streams

C++ provides a <sstream> header , which uses the same public interface to support I/O between a program and string object.

The string streams is based on **istringstream(subclass of istream)**, and **ostringstream(subclass of ostream) and bidirectional stringstream(subclass of iostream)**,

General Form:

```
typedef  basic_istringstream<char>istringstream;
typedef  basic_ostringstream<char>ostringstream;
```

Stream input can be used to validate input data,stream output can be used to format the output.

Ostringstream constructors

```
explicit ostringstream(ios::openmode mode=ios::out);//default with empty string
explicit ostringstream(const string &str, ios::openmode
mode=ios::out);//with initial str
string str() const;//get contents
void str(const string &s)//set contents
```

Example:

```
ostringstream sout;
//write into string buffer
sout<<"apple"<<endl;
sout<<"orange"<<endl;
//get contents
cout<<sout.str()<<endl;
```

ostringstream is responsible for dynamic memory allocation and management.

istringstream constructors

```
explicit istringstream(ios::openmode mode=ios::in); //default with empty string
explicit istringstream(const string &str, ios::openmode mode=ios::in); //with initial str
```

Example:

```
istringstream sin("123    12.34    hello");
//read from buffer
int I;
double d;
string s;
sin>>i>>d>>s;
cout<<i<<","<<d<<","<<s<<endl;
```

stringstream constructors

```
explicit stringstream(ios::openmode mode = ios::in | ios::out);
explicit stringstream(const string &str,
ios::openmode mode = ios::in | ios::out);
```

Program:

```
// Demonstrate string streams.
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main()
{
    stringstream s("This is initial string.");
    // get string
    string str = s.str();
    cout << str << endl;
    // output to string stream
    s << "Numbers: " << 10 << " " << 123.2;
    int i;
    double d;
    s >> str >> i >> d;
    cout << str << " " << i << " " << d;
    return 0;
}
```

Output:

```
This is initial string.
Numbers: 10 123.2
```

File streams

Formatted file streams

To perform file I/O, you must include the header **<fstream>** in your program. It defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. Remember, **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios**

Opening and Closing a File

open()

In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream.

There are three types of streams:

Input

To create an input stream, you must declare the stream to be of class **ifstream**.

Output

To create an output stream, you must declare it as class **ofstream**.

Input/Output

Streams that will be performing both input and output operations must be declared as class **fstream**.

General form for creating streams

```
    ifstream in; // input
    ofstream out; // output
    fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using **open( )**. This function is a member of each of the three stream classes.

Prototype:

void ifstream::open(const char **filename*, ios::openmode *mode* = ios::in);

void ofstream::open(const char **filename*, ios::openmode *mode* = ios::out | ios::trunc);

void fstream::open(const char **filename*, ios::openmode *mode* = ios::in | ios::out);

filename is the name of the file; it can include a path specifier. The value of *mode* determines how the file is opened. It must be one or more of the following values

- **ios::app :** Including **ios::app** causes all output to that file to be appended to the end. This value can be used only with files capable of output.
- **ios::ate :** Including **ios::ate** causes a seek to the end of the file to occur when the file is opened.
- **ios::in :** The **ios::in** value specifies that the file is capable of input.
- **ios::out :** The **ios::out** value specifies that the file is capable of output.
- **ios::binary :** The **ios::binary** value causes a file to be opened in binary mode. By default, all files are opened in text mode.
- **ios::trunc :** The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length.

Checking open() is successful or not

a. If **open( )** fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded.

Example:

```
 if(!mystream) {
cout << "Cannot open file.\n";
// handle error
}
```

b. You can also check to see if you have successfully opened a file by using the **is_open( )** function, which is a member of **fstream**, **ifstream**, and **ofstream**.

Prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise.

Example:

```
if(!mystream.is_open()) {
cout << "File is not open.\n";
// ...
```

close()

To close a file, use the member function **close( )**

Prototype: mystream.close();

The **close( )** function takes no parameters and returns no value.

Reading and Writing Text Files

It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using **cin** and **cout**, substitute a stream that is linked to a file.

Program:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ifstream in("INVNTRY"); // input
   if(!in)
   {
      cout << "Cannot open INVENTORY file.\n";
      return 1;
```

```
    }
    char item[20];
    float cost;
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in.close();

    ofstream out;
    out.open("INVNTRY");// output, normal file
    if(!out)
    {
        cout << "Cannot open INVENTORY file.\n";
        return 1;
    }
    out << "Radios " << 39.95 << endl;
    out << "Toasters " << 19.95 << endl;
    out << "Mixers " << 24.80 << endl;
    out.close();
    return 0;
}
```

Output:

```
Radios 39.95
Toasters 19.95
Mixers 24.8
```

Unformatted and Binary I/O

There will be times when you need to store unformatted (raw) binary data, not text. When performing binary operations on a file , openshould use **ios::binary** mode specifier

- **get( )**
  get( ) will read a character

General form: istream &get(char &*ch*);

reads a single character and puts that value in *ch*. It returns a reference to the stream

Overloading of get()

The **get( )** function is overloaded in several different ways.

Prototypes:

istream &get(char *buf*, streamsize *num*);

reads characters into the array pointed to by *buf* until either *num*-1

istream &get(char *buf*, streamsize *num*, char *delim*);

reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered.

int get( );

returns the next character from the stream

- **put( )**
  put( ) will write a character.

General form: ostream &put(char *ch*);
writes *ch* to the stream and returns a reference to the stream.

- **read( ) and write( )**
  Used to read and write blocks of binary data.

Prototypes:
istream &read(char **buf*, streamsize *num*);
reads *num* characters from the invoking stream and puts them in the buffer pointed to by *buf*.

ostream &write(const char **buf*, streamsize *num*);
writes *num* characters to the invoking stream from the buffer pointed to by *buf*.

- **getline( )**
  It also performs input. It is a member of each input stream class.

Prototypes:
istream &getline(char **buf*, streamsize *num*);
reads characters into the array pointed to by *buf* until either *num*-1

istream &getline(char **buf*, streamsize *num*, char *delim*);
reads characters into the array pointed to by *buf* until either *num*−1 characters have been read, the character specified by *delim* has been found

- **Detecting EOF**
  You can detect when the end of the file is reached by using the member function **eof( )**

Prototype: bool eof( );
It returns true when the end of the file has been reached; otherwise it returns false.

- **ignore( ) Function**
  You can use the **ignore( )** member function to read and discard characters from the input stream.

Prototype:
istream &ignore(streamsize *num*=1, int_type *delim*=EOF);
It reads and discards characters until either *num* characters have been ignored (1 by default) or the character specified by *delim* is encountered (**EOF** by default).

- **peek( )**
  You can obtain the next character in the input stream without removing it from that stream by using **peek( ).**

Prototype: int_type peek( );
It returns the next character in the stream or **EOF** if the end of the file is encountered.

- **putback( )**
  You can return the last character read from a stream to that stream by using **putback( ).**

Prototype : istream &putback(char *c*);
where *c* is the last character read.

- **flush( )**
  We can force the information to be physically written to disk before the buffer is full by calling **flush( )**.

Prototype: ostream &flush( );

Random Access

You perform random access by using the **seekg( )** and **seekp( ).**

- **seekg( )**
  The **seekg( )** function moves the associated file's current get pointer *offset* number

of characters from the specified *origin*, which must be one of these three values:

ios::beg	Beginning-of-file
ios::cur	Current location
ios::end	End-of-file

Prototype: istream &seekg(off_type *offset*, seekdir *origin*);

- **seekp( )**

The **seekp( )** function moves the associated file's current put pointer *offset* number of characters from the specified *origin*

Prototype: ostream &seekp(off_type *offset*, seekdir *origin*);

Obtaining the Current File Position

You can determine the current position of each file pointer by using these functions:

Prototypes: pos_type tellg( );
pos_type tellp( );

Here, **pos_type** is a type defined by **ios** that is capable of holding the largest value that either function can return.

You can use the values returned by **tellg( )** and **tellp( )** as arguments to the following forms of **seekg( )** and **seekp( )**, respectively.

istream &seekg(pos_type *pos)*;
ostream &seekp(pos_type *pos*);

Error handling during files operations

We have been opening and using the files for reading and writing on the assumption that everything is fine with the files. This may not be true always true.

For instance, one of the following things may happen when dealing with the files,

1. A file which we are attempting to open for reading does not exists
2. The file name used for a new file may already exists
3. We may attempt an invalid operation such as reading past the EOF.
4. There may not be any space in the disk for storing more data.
5. We may use an invalid file name.
6. We may attempt to perform an operation when the file is not opened for that purpose.

We can handle these types of error situations in the following ways,

a. I/O Status

The C++ I/O system maintains status information about the outcome of each I/O operation. The current state of the I/O system is held in an object of type **iostate**, which is an enumeration defined by **ios** that includes the following members.

Name	Meaning
ios::goodbit	No error bits set
ios::eofbit	1 when end-of-file is encountered; 0 otherwise
ios::failbit	1 when a (possibly) nonfatal I/O error has occurred;0 otherwise
ios::badbit	1 when a fatal I/O error has occurred; 0 otherwise

There are two ways in which you can obtain I/O status information.

- Call the **rdstate( )** function.

Prototype: iostate rdstate( );

It returns the current status of the error flags.

- We can determine if an error has occurred is by using one or more of these functions:

bool bad( );	The **bad( )** function returns true if **badbit** is set.
bool eof( );	returns true when end of the file has reached
bool fail( );	The **fail( )** returns true if **failbit** is set.

bool good( );	The **good( )** function returns true if there are no errors. Otherwise, it returns false.

Clearing an Error

Once an error has occurred, it may need to be cleared before your program continues. To do this, use the **clear( )** function.

Prototype: void clear(iostate *flags*=ios::goodbit);

If *flags* is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set *flags* as you desire.

Formatted I/O.

The C++ I/O system allows you to format I/O operations. There are two related but conceptually different ways that you can format data.

1. Directly access members of the **ios** class.(flags and functions in ios class)
2. Special functions called *manipulators*

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted.

a. Flags

The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

- When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.
- When the **left** flag is set, output is left justified.
- When **right** is set, output is right justified.
- When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character.
- **oct** flag causes output to be displayed in octal.
- Setting the **hex** flag causes output to be displayed in hexadecimal.
- To return output to decimal, set the **dec** flag.
- Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.
- By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.
- Setting **showpos** causes a leading plus sign to be displayed before positive values.
- Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.
- By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.
- When **unitbuf** is set, the buffer is flushed after each insertion operation.
- When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.
- Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**.
- Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**.
- Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the Format Flags

To set a flag, use the **setf( )** function. This function is a member of **ios**.

Common form: fmtflags setf(fmtflags *flags*);

This function returns the previous settings of the format flags and turns on those flags specified by *flags*.

Example: stream.setf(ios::showpos);

stream is the stream you wish to affect.

NOTE: The format flags are defined within the **ios** class, you must access their values by using **ios** and the scope resolution operator.

Clearing Format Flags

The complement of **setf( )** is **unsetf( )**. This member function of **ios** is used to clear one or more format flags.

General form: void unsetf(fmtflags *flags*);

The flags specified by *flags* are cleared.

Overloaded Form of setf( )

There is an overloaded form of **setf( ) .**

General form: fmtflags setf(fmtflags *flags1*, fmtflags *flags2*);

In this version, only the flags specified by *flags2* are affected. the most common use of the two-parameter form of **setf( )** is when setting the number base, justification, and format flags.

Program:
```
#include <iostream>
using namespace std;

int main ()
{
    cout.setf (ios::uppercase | ios::scientific);
    cout << 100.12;                  // displays 1.001200E+02
    cout.unsetf (ios::uppercase);  // clear uppercase
    cout << " \n" << 100.12 << endl;       // displays 1.001200e+02
    //OVERLOADED FORM OF setf
    cout.setf (ios::showpoint | ios::showpos, ios::showpoint);
    cout << 100.0<<endl;                   // displays 100.000, not +100.000
    //TWO PARAMETER FORM  of setf
    cout.setf(ios::hex, ios::basefield);
    cout << 100; // this displays 64

    return 0;
}
```

Output:
```
1.001200E+02
1.001200e+02
1.000000e+02
64
```

Setting All Flags

The **flags( )** function has a second form that allows you to set all format flags associated with a stream.

Prototype: fmtflags flags(fmtflags *f*);

When you use this version, the bit pattern found in *f* is used to set the format flags associated with the stream. Thus, all format flags are affected. The function returns the previous settings.

Example: cout.flags(f);

Program:
```
#include <iostream>
using namespace std;

void showflags();
int main()
{
    // show default condition of format flags
    showflags();
    // showpos, showbase, oct, right are on, others off
    ios::fmtflags f = ios::showpos | ios::showbase | ios::oct | ios::right;
    cout.flags(f); // set all flags
    showflags();
    return 0;
}
// This function displays the status of the format flags.
void showflags()
{
    ios::fmtflags f;
    long i;
    f = cout.flags(); // get flag settings
    // check each flag
    for(i=0x4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";
    cout << " \n";
}
```

Output:
```
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 1 0 1 0 1 1 0 0 0 0 0 0
```

b. Functions

There are three member functions defined by **ios.**

- **width( )**

By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width()** function.

Prototype; streamsize width(streamsize *w*);

Here, *w* becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The **streamsize** type is defined as some form of integer by the compiler.

After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width. If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.

- **precision( )**

When outputting floating-point values, you can determine the number of digits of precision by using the **precision( )** function.

Prototype: streamsize precision(streamsize *p*);

The precision is set to *p*, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If it is not, then the default precision will be used.

- **fill( )**

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill( )** function.

Prototype: char fill(char *ch*);

After a call to **fill( )**, *ch* becomes the new fill character, and the old one is returned.

Overloaded forms of width( ), precision( ), and fill( )

There are overloaded forms of **width( )**, **precision( )**, and **fill( )** that obtain but do not change the current setting. These forms are shown here:

```
char fill( );
streamsize width( );
streamsize precision( );
```

Program:

```
#include <iostream>
using namespace std;


int main ()
{
    cout.precision (4);
    cout.width (10);
    cout << 10.12345 << "\n";   // displays 10.12
    cout.fill ('*');
    cout.width (10);
    cout << 10.12345 << "\n";   // displays *****10.12
  // field width applies to strings, too
    cout.width (10);
    cout << "Hi!" << "\n";      // displays *******Hi!
    cout.width (10);
    cout.setf (ios::left);  // left justify
    cout << 10.12345;           // displays 10.12*****
    return 0;
}
```

Output:

```
     10.12
*****10.12
*******Hi!
10.12*****
```

Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. many of the I/O manipulators parallel member functions of the **ios** class.

Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolapha** flag.	Input/Output
dec	Turns on **dec** flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on **fixed** flag.	Output
flush	Flush a stream.	Output
hex	Turns on **hex** flag.	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
nobooalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output

noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program.

Program:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main ()
{
    cout << hex << 100 << endl;
    cout << setfill ('?') << setw (10) << 2343.0;
    return 0;
}
```

Output:

```
64
??????2343
```

Advantage

The main advantage of using manipulators instead of the **ios** member functions is that they often allow more compact code to be written. You can use the **setiosflags( )** manipulator to directly set the various format flags related to a stream.

Program:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main ()
{
    cout << setiosflags (ios::showpos);
    cout << setiosflags (ios::showbase);
    cout << 123 << " " << hex << 123;
    return 0;
}
```

```
+123 0x7b
```