Unit-IV

Pointer

A pointer is a variable that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to “point to” y. Pointer variables must be declared as such. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer’s base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable.

To use pointer:

- We define a pointer variable.
- Assign the address of a variable to a pointer.
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

  Example:

  ```
  int a=10;            //normal variable
  int*p;               //declare pointer
  p = &a;              // Assign the address of a variable “a” to a pointer “p”
  cout<<”a=”<<*p;      //prints a=10
  ```

OBJECT POINTERS

We have been accessing members of an object by using the dot operator. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed. We can declare an object pointer just as a pointer to any other type of variable is declared. Specify its class name, and then precede the variable name with an asterisk. To obtain the address of an object, precede the object with the & operator, just as you do when taking the address of any other type of variable.

Here is a simple example,

```
#include< iostream>
#include<conio.h>
class myclass {
int a;
public:
myclass(int x); //constructor
int get( );
};
myclass :: myclass(int x) {
```

```
a=x;
}
int myclass :: get( ) {
return a;
}
int main( ) {
myclass ob(120); //create object
myclass *p; //create pointer to object
p=&ob; //put address of ob into p
cout <<"value using object: " <<ob.get( );
cout <<"\n";
cout <<"value using pointer: " <<p->get( );
return0;
}
```

Notice how the declaration : myclass *p; creates a pointer to an object of myclass. It is important to understand that creation of an object pointer does not create an object. It creates just a pointer to one. The address of ob is put into p by using the statement:

p=&ob;

Finally, the program shows how the members of an object can be accessed through a pointer.

Pointers to Derived Types

Pointers to base classes and derived classes are related in ways that other types of pointers are not. In general, a pointer of one type cannot point to an object of another type. However, base class pointers and derived objects are the exceptions to this rule. In C++, a base class pointer can also be used to point to an object of any class derived from that base. For example, assume that you have a base class called B and a class called D, which is derived from B. Any pointer declared as a pointer to B can also be used to point to an object of type D. Therefore, given

```
B *p;           //pointer p to object of type B
B  B_ob;        //object of type B
D D_ob;         //object of type D
```

both of the following statements are perfectly valid:

```
p = &B_ob;      //p points to object B
p = &D_ob;      //p points to object D, which is an object derived from B
```

A base pointer can be used to access only those parts of a derived object that were inherited from the base class. Thus, in this example, p can be used to access all elements of D_ob inherited from B_ob. However, elements specific to D_ob cannot be accessed through p.

Another point to understand is that although a base pointer can be used to point to a derived object, the reverse is not true. That is, you cannot access an object of the base type by using a derived class pointer.

C++ Virtual Function

A virtual function is a member function that is declared within a base class and redefined by a derived class. In order to make a function virtual, you have to add keyword virtual in front of a function definition. When a class containing a virtual function is inherited, the derived class redefines the virtual function relative to the derived class. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method. When a virtual function is redefined by a derived class, the keyword virtual is not needed. A virtual function can be called just like any member function. However, what makes a virtual function interesting, and capable of supporting run-time polymorphism, is what happens when a virtual function is called through a pointer. When a base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function will be executed based upon the type of object being pointed to by the pointer. And this determination is made at run time. Therefore, if two or more different classes are derived from a base class that contains a virtual function, then when different objects are pointed to by a base pointer, different versions of the virtual function are executed.

```
// A simple example using a virtual function.
#include<iostream.h>
#include<conio.h>
class base {
public:
virtual void func( ) {
cout<< "Using base version of func(): ";
}
};
class derived1 : public base {
public:
voidfunc( ) {
cout<< "Using derived1's version of func(): ";
}
};
class derived2 : public base {
public:
voidfunc( ) {
```

```
cout<< "Using derived2's version of func(): ";
}
};
int main( ) {
base *p;
base ob;
derived1 d_ob1;
derived2 d_ob2;
p = &ob;
p- >func( );  // use base's func( )
p = &d_ob1;
p- >func( );  // use derived1's func( )
p = &d_ob2;
p- >func( ); // use derived2's func( )
return 0;
}
```

Pure virtual functions

Sometimes when a virtual function is declared in the base class, there is no meaningful operation for it to perform. This situation is common because often a base class does not define a complete class by itself. Instead, it simply supplies a core set of member functions and variables to which the derived class supplies the remainder. When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class must override this function. To ensure that this will occur, C++ supports pure virtual functions. A pure virtual function has no definition relative to the base class. Only the function prototype is included. To make a pure virtual function, use this general form:

virtual type func-name(parameter-list) = 0;

The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does not, a compile-time error results. Thus, making a virtual function pure is a way to guaranty that a derived class will provide its own redefinition.

Abstract class

When a class contains atleast one pure virtual function, it is referred to as an abstract class. Since, an abstract class contains atleast one function for which no body exists, it is, technically, an incomplete type, and no objects of that class can be created. Thus, abstract classes exist only to be inherited. It is important to understand, however, that you can still create a pointer to an abstract class, since it is through the use of base class pointers that run-time polymorphism is achieved. (It is also possible to have a reference to an abstract class.) .

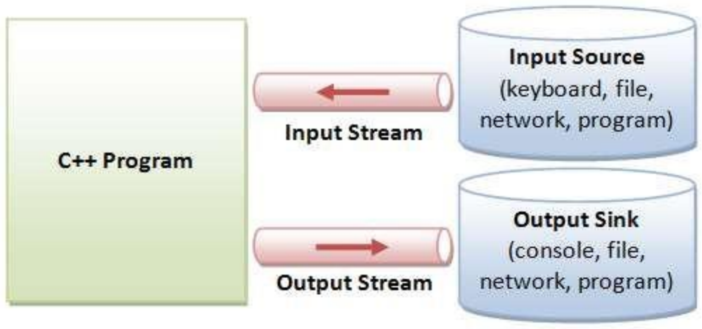
C++ Streams

The C++ I/O system operates through streams. A stream is logical device that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the I/O system presents the programmer with a consistent interface.

Two types of streams:

Output stream: a stream that takes data from the program and sends (writes) it to destination.

Input stream: a stream that extracts (reads) data from the source and sends it to the program.


C++ Program
Input Stream
Output Stream
Input Source
(keyboard, file, network, program)
Output Sink
(console, file, network, program)

C++ provides both the formatted and unformatted IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.

C++ provides supports for its I/O system in the header file< iostream>. Just as there are different kinds of I/O (for example, input, output, and file access), there are different classes depending on the type of I/O. The following are the most important stream classes:

Class istream :- Defines input streams that can be used to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions. Declares input functions such get(), getline() and read().

Class ostream :- Defines output streams that can be used to write data. Declares output functions put and write().The ostream class contains the overloaded insertion (<<) operator function

When a C++ program begins, these four streams are automatically opened:

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

Cin and Cout objects

cout is an object of class ostream. The cout is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents monitor. In the language of C++, the << operator is referred to as the insertion operator because it inserts data into a stream. It inserts or sends the contents of variable on its right to the object on its left.

For example:

cout << ``Programming in C++";

Here << operator is called the stream insertion operator and is used to push data into a stream (in this case the standard output stream)

cin is an object of class istream. cin is a predefined object that corresponds to the standard input stream. The standard input stream represents keyboard. The >> operator is called the extraction operator because it extracts data from a stream. It extracts or takes the value from the keyboard and assigns it to the variable on it's right.

For example:

int number;

cin >> number;

Here >> operator accepts value from keyboard and stores in variable number.

Unformatted Input/Output Functions

Functions get and put

The get function receives one character at a time. There are two prototypes available in C++ for get as given below:

get (char *)

get ()

Their usage will be clear from the example below:

char ch ;

cin.get (ch);

In the above, a single character typed on the keyboard will be received and stored in the character variable ch.

Let us now implement the get function using the other prototype:

char ch ;

ch = cin.get();

This is the difference in usage of the two prototypes of get functions.

The complement of get function for output is the put function of the ostream class. It also has two forms as given below:

cout.put (var);

Here the value of the variable var will be displayed in the console monitor. We can also display a specific character directly as given below:

cout.put (‘a’);

getline and write functions

C++ supports functions to read and write a line at one go. The getline() function will read one line at a time. The end of the line is recognized by a new line character, which is generated by pressing the Enter key. We can also specify the size of the line.

The prototype of the getline function is given below:

cin.getline (var, size);

When we invoke the above statement, the system will read a line of characters contained in variable var one at a time. The reading will stop when it encounters a new line character or when the required number (size-1) of characters have been read, whichever occurs earlier. The new line character will be received when we enter a line of size less than specified and press the Enter key. The Enter key or Return key generates a new line character. This character will be read by the function but converted into a NULL character and appended to the line of characters.

Similarly, the write function displays a line of given size. The prototype of the write function is given below:

write (var, size) ;

where var is the name of the string and size is an integer.

Formatted I/O via manipulators

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. I/O manipulators are special I/O format functions that can occur within an I/O statement.

Manipulator	Purpose	Input/Ouput
boolalpha	Turns on boolaphaflag	Input/Output
dec	Turns on decflag	Input/Output
endl	Outputs a newline character and flushes the stream	Output
ends	Outputs a null	Output
fixed	Turns on fixed flag	Output
flush	Flushes a stream	Output
hex	Turns on hexflag	Input/Output
internal	Turns on internalflag	Output
left	Turns on leftflag	Output
noboolalpha	Turns off boolalphaflag	Input/Output

noshowbase	Turns off showbaseflag	Output
noshowpoint	Turns off showpointflag	Output
noshowpos	Turns off showposflag	Output
noskipws	Turns off skipwsflag	Input
nounitbuf	Turns off unitbufflag	Output
nouppercase	Turns off uppercaseflag	Output
oct	Turns on octflag	Input/Output
resetiosflags(fmtflads f)	Turns off the flags specified in f	Input/Output
right	Turns on rightflag	Output
scientific	Turns on scientificflag	Output
setbase(int base)	Sets the number base to base	Input/Output
setfill(int ch)	Sets the fill char ch	Output
setiosflags(fmtflags f)	Turns on the flags specified by f	Input/Output
setprecision(int p)	Sets the number of digits of precision	Output
setw(int w)	Sets the field width to w	Output
showbase	Turns on showbaseflag	Output
showpoint	Turns on showpointflag	Output
showpos	Turns on showposflag	Output
skipws	Turns on skipwsflag	Input
unitbuf	Turns on unitbuf	Output
uppercase	Turns on uppercaseflag	Output
ws	Skips leading white space	Input

The following program demonstrates several of the I/O manipulators:

```
#include<iostream>
#include<iomanip>
using namespacestd;
int main( ) {
cout<< hex << 100 << endl;
cout<< oct<< 10 << endl;
cout<< setfill('X') << setw(10);
cout<< 100 << " hi " << endl;
return0;
}
```

This program displays the following:

```
64

13

XXXXXXX144 hi
```

File I/O

A file is a bunch of bytes stored on some storage devices like hard disk, floppy disk etc. File I/O and console I/O are closely related. In fact, the same class hierarchy that supports console I/O also supports the file I/O. To perform file I/O, you must include <fstream> in your program. It defines several classes, including ifstream, ofstream and fstream. In C++, a file is opened by linking it to a stream. There are three types of streams: input, output and input/output. Before you can open a file, you must first obtain a stream.

To create an input stream, declare an object of type ifstream.

To create an output stream, declare an object of type ofstream.

To create an input/output stream, declare an object of type fstream.

For example, this fragment creates one input stream, one output stream and one stream capable of both input and output:

```
ifstream in; // input;

fstream out;  // output;

fstream io;    // input and output
```

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

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

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

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

Here filename is the name of the file, which can include a path specifier. The value of the mode determines how the file is opened. It must be a value of type open mode, which is an enumeration defined by ios that contains the following value:

- ios::app: causes all output to that file to be appended to the end. Only with files capable of output.
- ios::ate: causes a seek to the end of the file to occur when the file is opened.
- ios::out: specify that the file is capable of output.
- ios::in: specify that the file is capable of input.
- ios::binary: causes the file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations might take place, such as carriage return/linefeed

sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur.

• ios::trunc: causes the contents of a pre-existing file by the same name to be destroyed and the file to be truncated to zero length. When you create an output stream using ofstream, any pre-existing file is automatically truncated.

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

ofstream myfile;

myfile.open ("example.bin", ios::out | ios::app | ios::binary);
Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

Class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Closing file

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function `close`. This member function takes flushes the associated buffers and closes the file:

myfile.close();

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes. In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

To write to a file, you construct a `ofsteam` object connecting to the output file, and use the `ostream` functions such as stream insertion `<<`, `put()` and `write()`. Similarly, to read from an input file, construct an `ifstream` object connecting to the input file, and use the `istream` functions such as stream extraction `>>`, `get()`, `getline()` and `read()`.

There are two ways of storing data in a file as given below:

Binary form and Text form

Suppose, we want to store a five digit number say 19876 in the text form, then it will be stored as five characters. Each character occupies one byte, which means that we will require five bytes to store five-digit integer in the text form. This requires storage of 40 bits. Therefore, if we can store them in binary form, then we will need only two bytes or 16 bits to store the number. The savings will be much more when we deal with floating point numbers.

When we store a number in text form, we convert the number to characters. However, storing a number in binary form requires storing it in bits. However, for a character, the binary representation as well as the text representation are one and the same since, in either case, it occupies eight bits. The text format is easy to read. We can even use a notepad to read and edit a text file. The portability of text file is also assured. In case of numbers, binary form is more appropriate. It also occupies lesser space when we store it in binary form and hence it will be faster. The default mode is text.

Unformatted, binary I/O

C++ supports a wide range of unformatted file I/O functions. The unformatted functions give you detailed control over how files are written and read. The lowest-level unformatted I/O functions are get( )and put( ). You can read a byte by using get( ) and write a byte by using put( ). These functions are member functions of all input and output stream classes, respectively. The get( )function has many forms, but the most commonly used version is shown here, along with put( ):

istream &get(char&ch);

ostream &put(char&ch);

To read and write blocks of data, use read( )and write()functions, which are also member functions of the input and output stream classes, respectively. Their prototypes are:

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

ostream &write(const char*buf, streamsize num);

The read( ) function reads num bytes from the stream and puts them in the buffer pointed to by buf. The write() function writes num bytes to the associated stream from the buffer pointed by buf. The streamsize type is some form of integer. An object of type streamsize is capable of holding the largest number of bytes that will be transferred in any I/O operation. If the end of file is reached before num characters have been read, read( ) stops and the buffer contains as many characters as were available.

When you are using the unformatted file functions, most often you will open a file for binary rather than text operations. The reason for this is easy to understand: specifying ios::binary prevents any character translations from occurring. This is important when the binary representations of data such as integers, floats and pointers are stored in the file. However, it is perfectly acceptable to use the unformatted functions on a file opened in text mode, as long as that the file actually contains only text. But remember, some character translation may occur.

Random Access

File pointers

C++ also supports file pointers. A file pointer points to a data element such as character in the file. The pointers are helpful in lower level operations in files. There are two types of pointers:

get pointer

put pointer

The get pointer is also called input pointer. When we open a file for reading, we can use the get pointer. The put pointer is also called output pointer. When we open a file for writing, we can use put pointer. These pointers are helpful in navigation through a file. When we open a file for reading, the get pointer will be at location zero and not 1. The bytes in the file are numbered from zero. Therefore, automatically when we assign an object to ifstream and then initialize the object with a file name, the get pointer will be ready to read the contents from 0th position. Similarly, when we want to write we will assign to an ofstream object a filename. Then, the put pointer will point to the 0th position of the given file name after it is created. When we open a file for appending, the put pointer will point to the

0th position. But, when we say write, then the pointer will advance to one position after the last character in the file.

File pointer functions

There are essentially four functions, which help us to navigate the file as given below Functions
Function Purpose

tellg()	Returns the current position of the get pointer
seekg()	Moves the get pointer to the specified location
tellp()	Returns the current position of the put pointer
seekp()	Moves the put pointer to the specified location

```
//To demonstrate writing and reading- using open
#include<fstream.>
#include<iostream>
int main(){                    //Writing
ofstream outf;
outf.open("Temp2.txt");
outf<<"Working with files is fun\n";
outf<<"Writing to files is also fun\n";
outf.close();
char buff[80];
ifstream inf;
inf.open("Temp2.txt");                    //Reading
while(inf){
inf.getline(buff, 80);
cout<<buff<<"\n";
}
inf.close();
return 0;
}
```