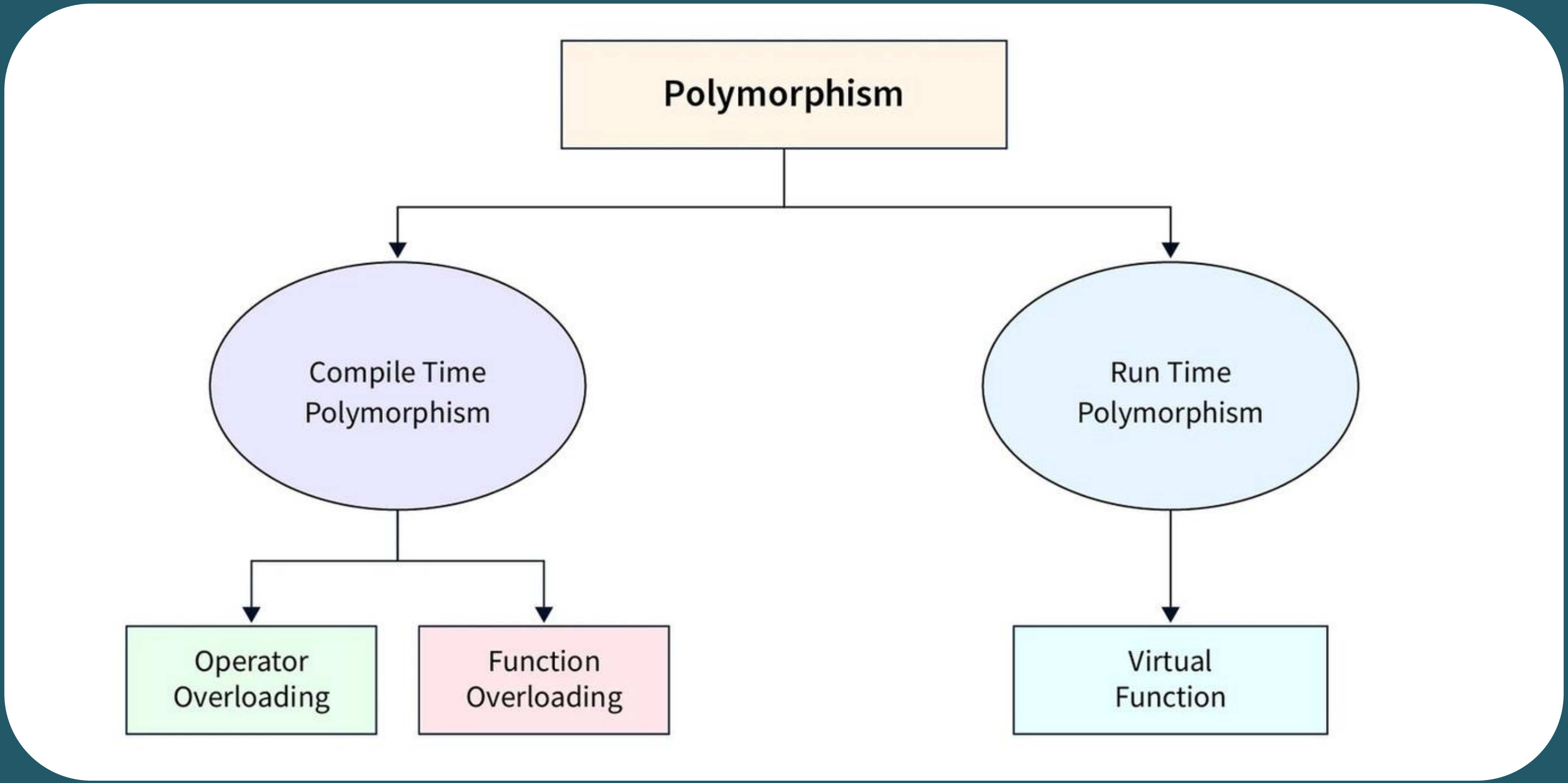
Polymorphism
&
Virutal Function

INTRODUCTION

Polymorphism is one of the crucial features of OOP which simply means " one name, multiple forms". We have already seen how the concept of polymorphism is implemented in the function overloading and operator overloading. There are two types of polymorphism namely, **compile time polymorphism** and **run-time polymorphism**.



Compile Time Polymorphism

At compile time, the compiler at the compile time knows all the matching arguments, therefore the compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking. Also known as compile time polymorphism. Early binding simply means that an object is bound to its function call at compile time.

Run Time Polymorphism

If appropriate member functions are chosen at run time rather than compile time, this is known as runtime polymorphism or late binding or runtime binding or dynamic binding. Dynamic binding(also known as runtime binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. Since an appropriate function is called during the runtime it needs the use of a pointer by making base class pointer points at different objects even that of derived class we can execute different versions of virtual function. It is associated with polymorphism and inheritance.

NEED OF VIRTUAL FUNCTIONS

```
#include<iostream.h>
#include<conio.h>
class base
{
    public: void
    show() {
      cout<<"you have called base class function";
        }
};

Class derived: public base
{

    public: void
    show()
    {
        cout<<"you have called derived class function";
    }
};
```

```
int main()
{
    base *b;
    derived d;
    b=&d;//base pointer to derived object
    b->show();
    getch();
    return 0;
}
```

Output:

you have called base class function

In the above program, even though the base pointer points to a derived class object, it cannot access the unique public function of the derived class.

This is because **the compiler ignores the content of the pointer and chooses a member function that matches the type of pointer**.

Here, *b is a pointer of base class "base" so b->show() only invokes the base class member function even though it is pointing to the derived class object d.

In order to resolve this issue, we need a virtual function.

Here, A virtual function can be used to call appropriate derived class functions by using keyword virtual.

Virtual Function

A C++ virtual function is a member function in the base class and is re-defined(overridden) in a derived class. It is declared using the virtual keyword. It is used to tell the compiler to perform runtime polymorphism or dynamic linkage or late binding on the function.

Syntax:

virtual void func_name() { }   *OR,*

void virtual func_name() { }

When there is a necessity to use the single pointer to refer to all the objects of the different classes, we create the pointer to the base class that refers to all the derived objects. But, when the base class pointer contains the address of the derived class object, it always executes the base class function. This issue can only be resolved by using the 'virtual' function. A 'virtual' is a keyword preceding the normal declaration of a function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Virtual Function

```
#include<iostream.h>
#include<conio.h>
class Base {
public:
virtual void show()
{cout<<"you have called Base class function";}
};
class Derived1:public Base {
public:
void show()
{
cout<<"you have called derived 1 class function"<<endl;}
 };
class Derived2:public Base {
public:
void show()
{ cout<<"you have called derived 2 class function"<<endl;
}};
```

```
int  main()  {
Base        *b;
Derived1 d1;
Derived2 d2;
b=&d1;
b->show();
b=&d2;
b->show();
getch();
return 0;
}
```

Output:

you have called derived 1 class function

you have called drived 2 class function

Virtual Function

In the above program, base class pointer b is pointing to the objects d1 and d2 of classes Derived1 and Derived2 respectively.

when b->show() is evaluated it actually refers to the show function of Base class i.e. Base::show(); but this show() function is a virtual function and it instructs the compiler to go and see another derived version of this function.

According to the object pointed by the base pointer, it looks into those derived classes and calls the appropriate function.

In the first case, the base pointer points to object ‘d1’ of Derived1 class so it calls show() function of Derived1 class and in second case, the base pointer points to object ‘d2’ of Derived2 class so calls the show() function of Derived2 class.

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASS

A **pure virtual function** (or abstract function) in C++ is a virtual function in the base class for which there is no implementation(no body).

A pure virtual function is one with an initialize of = 0 in its declaration. All the derived classes must override the base class pure virtual function and provide implementations of those functions. A class containing pure virtual functions cannot be used to declare an object of its own. such classes are called **abstract classes**. The difference between virtual function and pure virtual function is that a virtual function has an implementation and gives derived class an option of overriding the virtual functions whereas the pure virtual function doesn't provide implementation and requires the derived class to override those functions.

```
class Test        // An abstract class
{
            // Data members of class
    public:
            virtual void show() = 0;     // Pure Virtual Function
            /* Other members */
};
```

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASS

- A class containing at least one pure virtual function is known as an abstract class.
- We cannot create objects of abstract classes. But, we can create pointers for abstract classes which is required for runtime polymorphism.
- We must override the pure virtual function of an abstract class in the derived class, otherwise, the derived class will also become an abstract class.
- Normally, when creating a class hierarchy with virtual functions, in most of the cases it seems that the base class pointers are used but the objects of base class are rarely created. The concept of abstract classes seems useful in such scenarios.
- Abstract classes mostly exist to act as a parent of the derived class.

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASS

```
// C++ Program to illustrate the abstract class and
//virtual functions
#include <iostream>
using namespace std;
class Base
{
    protected:
    int x;
    public:
        virtual void sum() = 0; // pure virtual function
        virtual void input()
        {
            cout<<"Enter the value of x: ";
            cin>>x;
        }
};
```

```
class Derived: public Base
{
    int y;
    public:
        void input() {
            Base::input();
            cout<<"Enter the value of y: ";
            cin>>y;
            }
    void sum() { //overriding pure virtual function
     cout << "The sum is: "<< x+y<<endl; }
};
int main()
{
    Derived d;
    Base *bptr;
    bptr = &d;
    bptr -> input();
    bptr -> sum();
     return 0;
}
```

Output:

Enter the value of x: 5

Enter the value of y: 4

The sum is: 11

VIRTUAL DESTRUCTORS

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

```
// CPP program without virtual destructor causing undefined behavior
#include <iostream>
using namespace std;
class Base
{
public:
      Base()
     {
     cout << "Constructing base\n";
     }
     ~Base()  {
     cout<< "Destructing base\n";
     }
     };
```

```
class Derived: public Base {
 public:
        Derived()
 { cout << "Constructing derived\n"; }
 ~Derived()  {
        cout << "Destructing derived\n";
}
};
```

VIRTUAL DESTRUCTORS

```
int main()
{
Base * b = new Base;                //calls Base constructor
delete b;                           //calls Base destructor
cout<<”------------------------\n”;
Derived * d = new Derived;          // calls Derived constructor
delete d;                           //calls Derived destructor
cout<<”------------------------\n”;

Base * b = new Derived;             // calls Derived Destructor
delete b;                           // calls Base Destructors
getch();
return 0;
}
```

Output:

```
Constructing base
Destructing base
-----------------------------
Constructing base
Constructing derived
Destructing derived
Destructing base
-----------------------------
Constructing base
Constructing derived
Destructing base
```

VIRTUAL DESTRUCTORS

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.

```
// A program with virtual destructor
#include <iostream>
using namespace std;
class Base {
public:
Base()
{ cout << "Constructing base\n"; }
 virtual ~Base()
{ cout << "Destructing base\n"; } };
```

```
class Derived : public Base {
public:
Derived()
{cout << "Constructing derived\n"; }
~Derived()
{ cout << "Destructing derived\n"; }
};
```

```
int main() {
Base *b = new Derived;
delete b;
getch();
return 0;
 }
```

Output:

Constructing base
Constructing derived
Destructing derived
Destructing base

Virtual destructors are useful when you might potentially delete an instance of a derived class through a pointer to base class. As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing).

Multiple Choice Questions on Virtual Function

1. Which is used to create a pure virtual function?
a) $
b) =0
c) &
d) !

2. Which is also called as abstract class?
a) virtual function
b) pure virtual function
c) derived class
d) base class

3.Where does the abstract class is used?
a) base class only
b) derived class
c) both derived & base class
d) virtual class

4.Pick out the correct option.
a) We cannot make an instance of an abstract base class
b) We can make an instance of an abstract base class
c) We can make an instance of an abstract super class
d) We can make an instance of an abstract derived class

5. What is meant by pure virtual function?
a) Function which does not have definition of its own
b) Function which does have definition of its own
c) Function which does not have any return type
d) Function which does not have any return type & own definition

6. Which is the correct syntax of defining a pure virtual function?
a) pure virtual return_type func();
b) virtual return_type func() pure;
c) virtual return_type func() = 0;
d) virtual return_type func();

7. Pick the correct statement.
a) Pure virtual functions and virtual functions are the same
b) Both Pure virtual function and virtual function have an implementation in the base class
c) Pure virtual function has no implementation in the base class whereas virtual function may have an implementation in the base class
d) The base class has no pure virtual function

8. Which is the correct statement about pure virtual functions?
a) They should be defined inside a base class
b) Pure keyword should be used to declare a pure virtual function
c) Pure virtual function is implemented in derived classes
d) Pure virtual function cannot implemented in derived classes

9. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class A{
        int a;
    public:
        virtual void func() = 0;
};
class B: public A{
   public:
        void func()
            {    cout<<"Class B"<<endl;      }
};
int main(){
        B b;
        b.func();
        return 0;
}
```

a) Class B
b) Error
c) Segmentation fault
d) No output

10. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class A{
        int a;
    public:
        virtual void func() = 0;
};
class B: public A{
   public:
        void func()
            {    cout<<"Class B"<<endl;      }
};
int main(){
        A a;
        a.func();
        return 0;
}
```

a) Class B
b) Error
c) Segmentation fault
d) No output

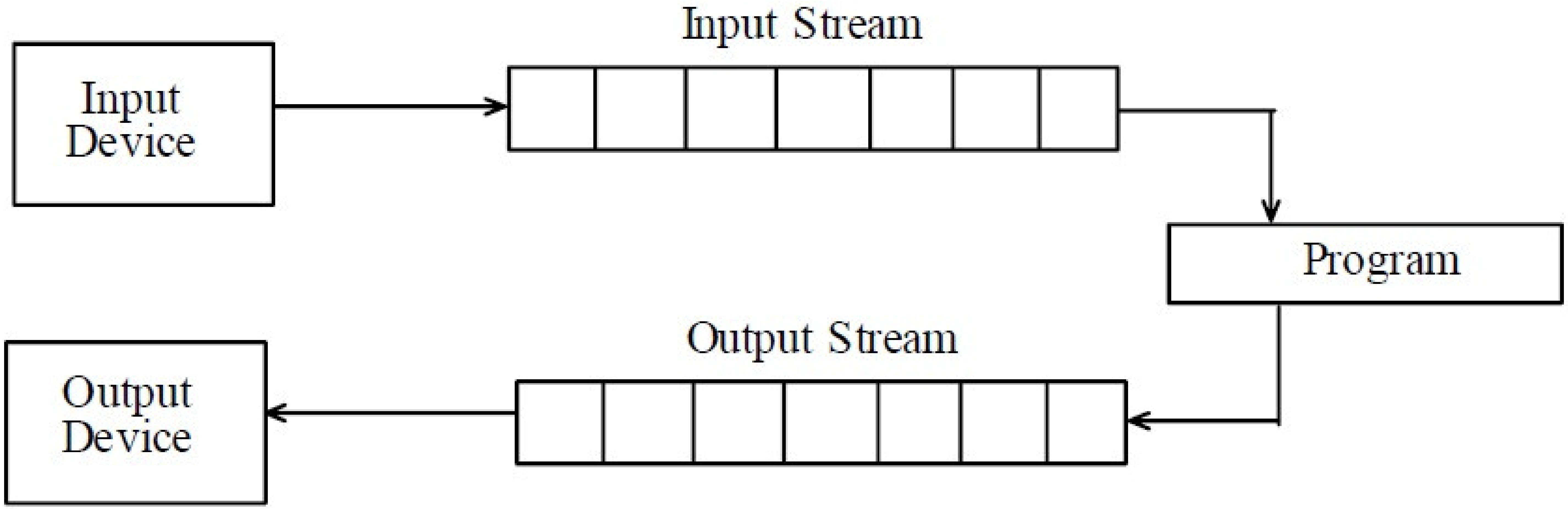
File Handling

STREAM INPUT/OUTPUT

A stream is an interface provided by I/O system to the programmer. A stream, in general, is a name given to flow of data. In other words, it is a sequence of bytes. The stream acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called **INPUT stream** and the destination stream that receives output from the program is called **OUTPUT stream**. Thus, a program extracts the bytes from an input stream and inserts them into the output stream.

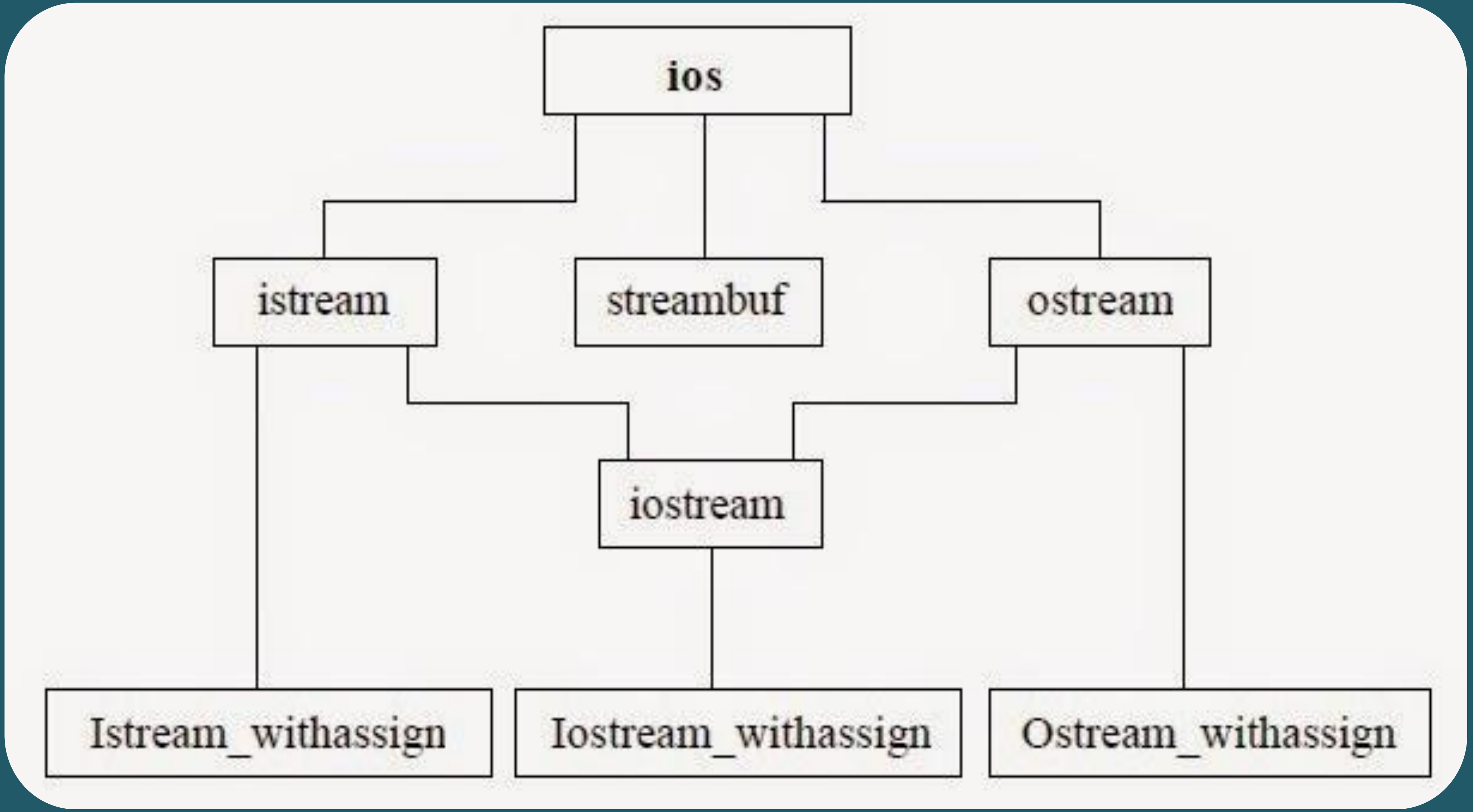
Different input devices like keyboard can send data to the input stream. Also, data in the output stream can go to the output device like screen (monitor) or any other storage device. In C++, there are predefined I/O stream like cin and cout which are automatically opened where a program begins its execution.

STREAM INPUT/OUTPUT



STREAM CLASS HIERARCHY FOR CONSOLE I/O

The given figure shows a hierarchy of stream classes in C++ used to define various stream in order to deal with both the console and disk files. From the figure, it is clear that ios is a base class. This ios class is declared as virtual base class so that only one copy of its members is inherited by its derived classes: thereby avoiding ambiguity.



The ios class comprises basic functions and constants required for input and output operations. It also comprises functions related with flag strings.

STREAM CLASS HIERARCHY FOR CONSOLE I/O

istream and **ostream** classes are derived from **ios** and are dedicated to input and output streams respectively. Their member functions perform both formatted and unformatted operations. **istream** contains functions like get( ), getline, read( ) and overloaded extraction(>>) operators. ostream comprises functions like put( ), write( ) and overloaded insertion(<<) operators. The **iostream** class is derived from **istream** and **ostream** using multiple inheritance. Thus, it provides the facilities for handling both input and output streams. The class **istream_withassign** and **ostream_withassign** add assignment operator to these classes. Again the classes **ifstream** and **ofstream** are concerned with file I/O function. **ifstream** is used for input files and ofstream for output files. Also there is another class of stream which will be used both for input and output. All the classes **ifstream**, **ofstream** and **fstream** are derived from classes **istream**, **ostream** and **iostream** respectively. **streambuf** is also derived from **ios** base class. **filebuf** is derived from **streambuf**. It is used to set the file buffers to read and write. It also contains open( ) and close( ) used to open and close the file respectively.

UNFORMATTED INPUT/OUTPUT

1. Overloaded Operators >> and <<:

The general format for reading data from keyboard is :

cin>>var1>>var2>>......>>varn.

This statement will cause the computer to stop the execution and look for input data from the keyboard. While entering the data from the keyboard, the whitespace, newline and tabs will be skipped. The operator >> reads the data character by character basis and assigns it to the indicated locations. Again, the general format for displaying data on the computer screen is :

cout<<item1<<item2<<..........<<itemn;

Here, item1, item2, ..., itemn may be character or variable of any built-in data type.

2. get() and put():

These are another kind of input/output functions defined in classes istream and ostream to perform single character input/output operations.

get():

There are 2 types of get functions i.e. get(char*) and get(void) which help to fetch a character including the blank space, tab and a new line character. get(char) assigns input character to its argument and get(void) returns the input character.

```
char c;
cin.get(c) ; // obtain single character from keyboard and assign it to char c
OR
c=cin.get( ) ;
OR
cin>>c ;        // this will skip white spaces, tabs, and newline
```

put():

It is used to output a line of text character by character basis.

```
cout.put ('T') ; // prints T
cout.put(ch) ; // prints the value of variable ch
cout.put(65) ; // displays a character whose ASCII value is 65 that is A
```

3. getline() and write():

getline():

The getline( ) function reads a whole line of text that ends with a newline character.

The general syntax is:

cin.getline(line, size) ;

where, line is a variable, size is maximum number of characters to be placed.

Consider the following code:

```
char name[30] ;
cin.getline(name, 30)
or
cin>>name
```

```
string name;
getline(cin,name); // in case of variable declared as string
```

If we input the following string from keyboard:
“This is test string”.
cin.getline(name, 30) inputs 29 characters taking white spaces and one left null character. so, it will take the whole line but in case of cin it takes only “This” as it doesn’t consider white spaces.

Example:

```
#include <iostream>

using namespace std;
int main( )

{
    char city[20] ;

    cout<< “Enter city name:\n” ;
    cin>>city ;

    cout<< “city name:”<<city<<”\n\n” ;
    cout<< “Enter city name again:\n” ;

    cin.getline (city, 20) ;
    cout<< “New city name:”<<city<< “\n\n” ;

}
```

Output:

First Run:
Enter city name: Kathmandu

City name : Kathmandu
Enter city name again : Lalitpur

New city name : Lalitpur

Second Run:
Enter city name : New Baneshwor

City name : New
Enter city name again : Old Baneshwor

New city name : Baneshwor

write():

This function displays an entire line of text in the output string.

The general syntax is:

cout.write(line, size)

where, line represents the name of string to be displayed and second argument size indicates number of characters to be displayed.

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

int main( )
{

 Char str[30] = “HELLO WELCOME TO KEC”;
 cout.write(str, 30);

 cout.write( str, 8);
}
```

Output:

HELLO WELCOME TO KEC

HELLO WE

FORMATTED INPUT/OUTPUT

C++ supports a number of features that could be used for formatting the output. These features include

- ios class functions and flags
- manipulators

ios class functions and flags

width( ) - To specify the required field size for displaying an output value

Precision( ) - To specify the no. of digits to be displayed before and after a decimal point of a float value.

fill( ) - To specify a character that is used to fill the unused portion of a field

setf( ) - To specify format flags that can control the form of output display (such as left- left-justification and right-justification)

unsetf( ) - To clear flags specified

FORMATTED INPUT/OUTPUT

1. width():

This function of ios class is used to define the width of the field to be used while displaying the output. It is normally accessed with a cout object. It has the following form:

cout.width(6); *//sets field width to 6*

Example:

```
cout.width(6);
cout<<849<<endl;
cout<<45<<endl;
```

Output:

_ _ _849

45

The value 849 is printed right-justified in the first six columns. The specification width(6) does not retain the setting for printing the number 45.

This can be improved as follows:

```
cout.width(6);
cout<<849<<endl;
cout.width(6);
cout<<45<<endl;
```

Output:

---849

----45

FORMATTED INPUT/OUTPUT

2. fill():

This ios function is used to specify the character to be displayed in the unused portion of the display width. By default, blank characters are displayed in the unused portion.

Syntax:

cout.fill(ch);

where ch is a character used to fill the unused space

Example:

```
int x = 456;
cout.width(6);
cout.fill(‘#’);
cout<<x<<endl;
```

Output:

```
###456
```

FORMATTED INPUT/OUTPUT

3. precision():

This function belonging to ios class is used to specify maximum number of digits to be displayed as a while in floating point number or the maximum number of digits to be displayed in the fractional part of the floating point number. In general format, it specifies the maximum number of digits including fractional or integer parts. This is because in general format the system chooses either exponential or normal floating point format which best preserves the value in the space available.

cout.precision(4);

Example:

```
float x=5.5005, y=66.769;
cout.precision(3);
cout<<x<<endl;
cout<<y<<endl;
```

Output:

```
5.5
66.8
```

FORMATTED INPUT/OUTPUT

4. setf():

The ios member function setf() is used to set flags and bit fields that controls the output in other ways.

cout.setf(flag_value, bit_field_value);

Example:

```
int x = 456;
float y = 123.45;
cout.setf(ios::left, ios::adjustfield);
cout.width(6);
cout.fill('#');
cout<<x<<endl;
cout.setf(ios::scientific, ios::floatfield);
cout<<y<<endl
```

Flag_value	**bit_field_value**
ios::left ios::right ios::internal	ios::adjustfiled
ios::scientific ios::fixed	ios::floatfield
ios::dec ios::oct ios::hex	ios::basefield

Output:

```
456###
1.234500e+02
```

FORMATTING WITH MANIPULATORS

The header file “iomanip” provides a set of functions called ‘manipulators’ which can be used to manipulate the output formats. They provide the same features as that of the ios member functions and flags. We can use two or more manipulators as a chain in one statement

cout<<manip1<<manip2<<item;
cout<<manip1<<manip2<<item<<manip3;

Non-parameterized manipulators

endl: output new line and flush
left: sets ios::left flag of ios::adjustfield
right: sets ios::right flag of ios::adjustfield
dec: sets ios::dec flag of ios::basefield sets
Hex: ios::hex flag of ios::basefield
Oct: sets ios::oct flag of ios::basefield
showpoint: sets ios::showpoint flag
scientific: sets ios::scientific flag of ios::floatfield
fixed: sets ios::fixed flag of ios::floatfield

Parameterized manipulators

setw(int n) : equivalent to ios function width()

setprecision(int n) : equivalent to ios function precision()

setfill(char c) : equivalent to ios function fill()

setiosflags(flag) : equivalent to ios function setf()

resetiosflags(flag) : equivalent to ios function unsetf()

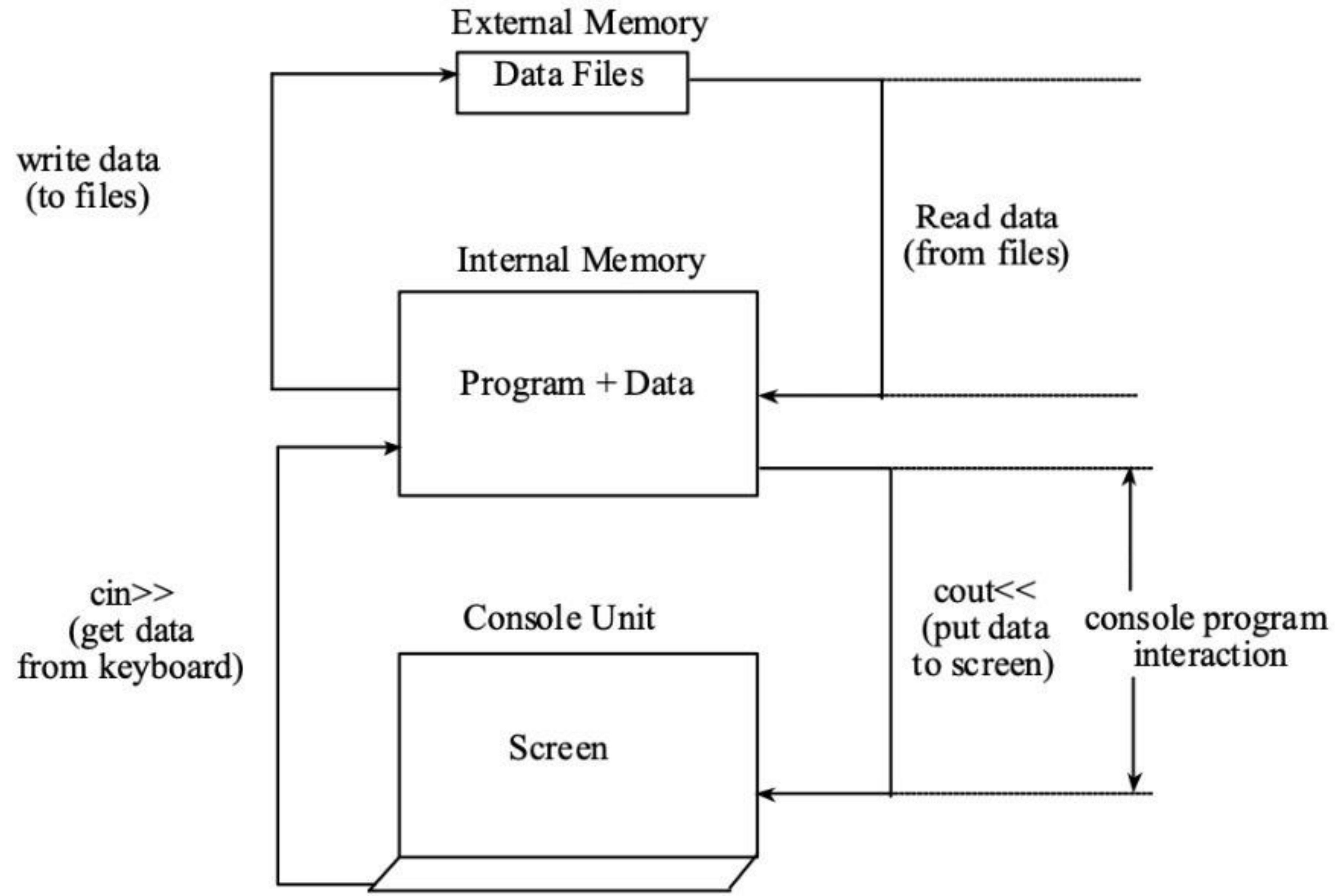
FILE INPUT/OUTPUT WITH STREAMS

All the programs presented so far take input from standard input device(normally keyboard) and output displayed on standard output device(normally monitor). The console stream objects like cout and cin have been used for output and input respectively. However, many applications may require a large amount of data to be read, processed, and also saved for later use. In order to handle such a huge volume of data, we need to use some devices such as floppy disks or hard disks to store the data. The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both the following kinds of data communication:

1. Data transfer between the console unit and the program
2. Data transfer between the program and a disk file.

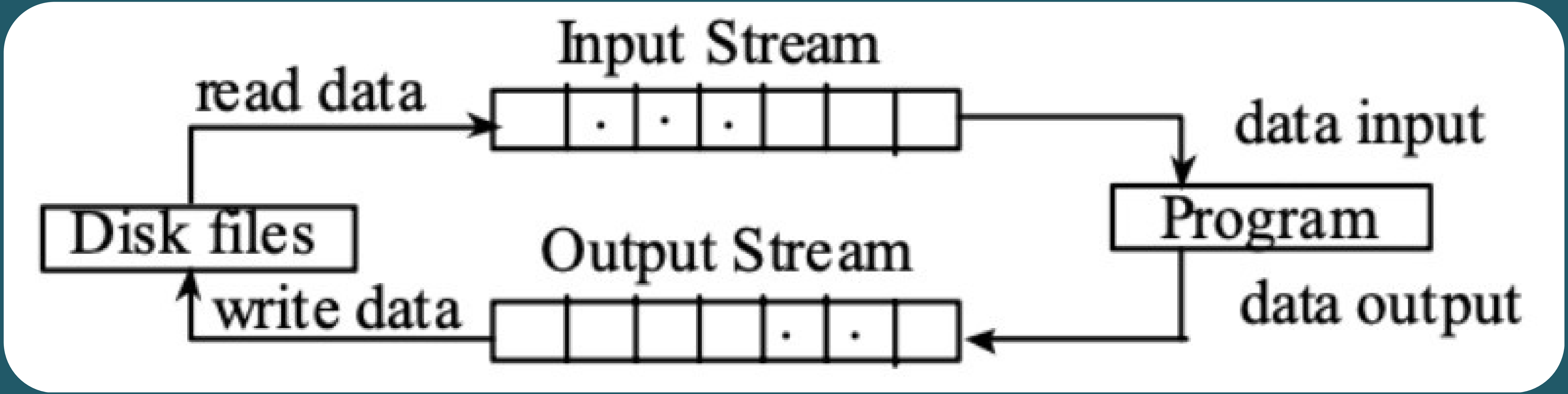
FILE INPUT/OUTPUT WITH STREAMS



FILE INPUT/OUTPUT WITH STREAMS

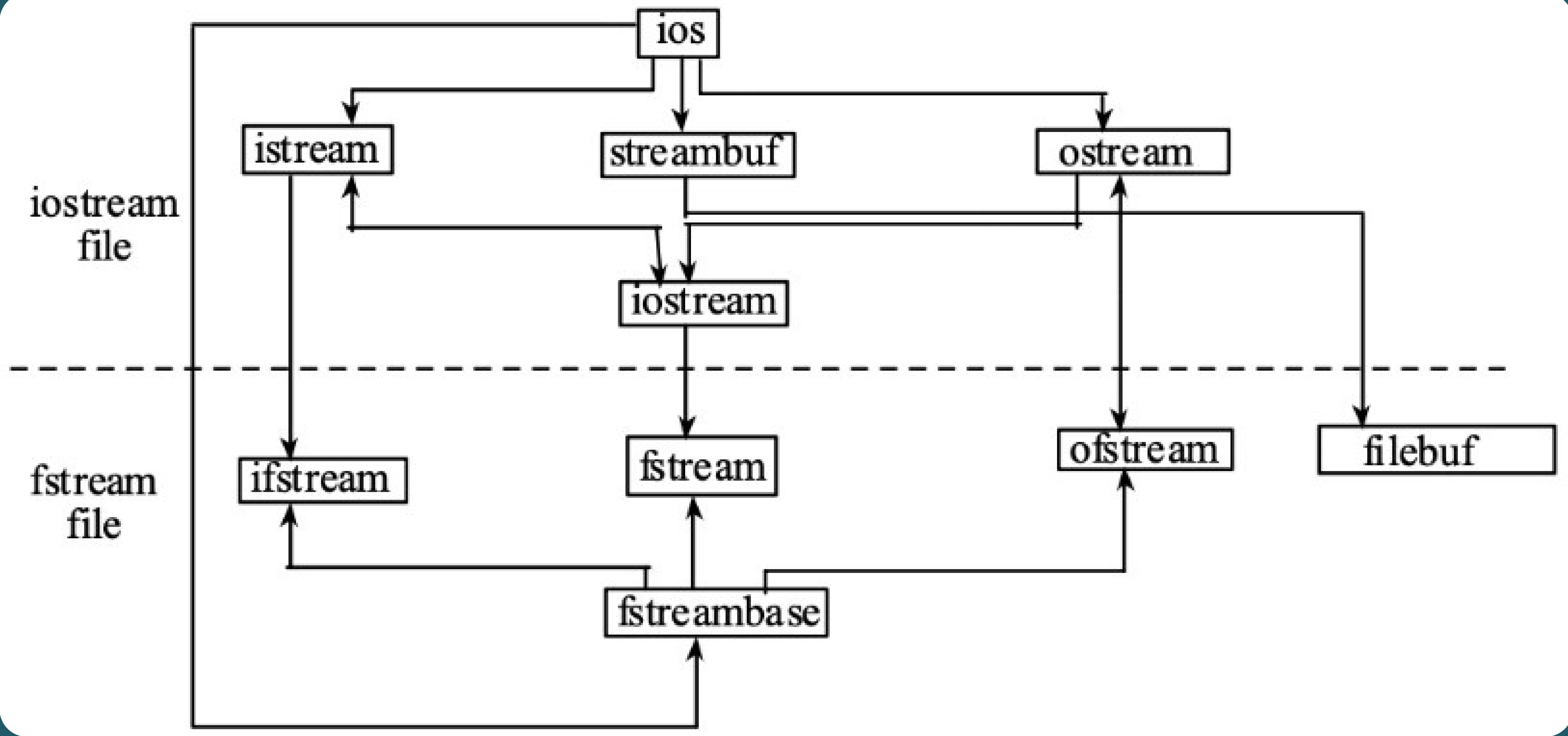
File Stream

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. A file stream is an interface between the programs and the files. The stream which supplies data to the program is called input stream and that which receives data from the program is called output stream i.e. the input stream reads or receives data from the file and supplies it to the program while the output stream writes or inserts data to the file.



FILE INPUT/OUTPUT WITH STREAMS

Class Hierarchy for File Stream



FILE INPUT/OUTPUT WITH STREAMS

Class Hierarchy for File Stream

The I/O system of C++ contains a set of classes that define the file handling methods. These include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream class as shown in figure. These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.

ios: It stands for input output stream. This class is the base class for other classes in this class hierarchy. This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

istream: istream stands for input stream. This class is derived from the class ‘ios’. This class handles input stream. The extraction operator(>>) is overloaded in this class to handle input operations from files to the program. This class declared input functions such as get(), getline() and read().

ostream: ostream stands for output stream. This class is derived from the class ‘ios’. This class handles output stream.The insertion operator(<<) is overloaded in this class to handle output streams to files from the program. This class declares output functions such as put() and write().

Class Hierarchy for File Stream

streambuf: This class contains a pointer which points to the buffer which is used to manage the input and output streams.

fstreambase: This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class. This class contains open() and close() function.

ifstream: This class provides input operations. It contains an open() function with default input mode. It inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

ofstream: This class provides output operations. It contains an open() function with default output mode. It inherits the functions put(), write(), seekp() and tellp() functions from the ostream.

fstream: This class provides support for simultaneous input and output operations. It inherits all the functions from istream and ostream classes through iostream.

filebuf: Its purpose is to set the file buffers to read and write. We can also use the file buffer member function to determine the length of the file.

OPENING AND CLOSING FILES

A file can be opened in two ways:

1. Using the constructor function of the class: This method is useful when we use only one file in the stream.
2. Using the member function open() of the class: This method is used when we wat to manage multiple files using the stream.

Opening Files using Constructor:

We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class.(.i.e. If we are writing to a file we create an object of class ofstream and if we are reading from the file, we create an object of the class ifstream).
2. Initialize the file object with the desired filename.

For example, the following statement opens a file named results for the output:

ofstream outfile("results"); *//output only*

This creates outfile as an object of the class ofstream that manages the output stream. This object can be any valid c++ name such as o_file, myfile, fout etc. This statement opens a file named "results" and attaches it to the output stream **outfile.**

OPENING AND CLOSING FILES

Opening Files using open():

As stated earlier, the function open() can be used to open multiple files that use the same stream object. For example,, we may want to process a set of file sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

file-stream-class stream-object;
stream-object.open("Filename);

Example:

```
ofstream outfile;        //create stream(for output)
outfile.open("Data1"); //connect output stream (outfile) to Data1 File.
.............
outfile.close();    //disconnect output stream(outfile) from Data1 File
outfile.open("Data2"); //connect output stream (outfile) to Data2 File.
.............
outfile.close();       //disconnect output stream(outfile) from Data2 File
.............
```

OPENING AND CLOSING FILES

Opening Files using open():

Here, we are opening the file Data1 and Data2 using the object “outfile” of the ofstream class. This means that we are only allowed to perform write operations in multiple files using the same output stream object “outfile”.

Suppose we want to perform both reading and writing operations on the same file, then we need to create an object of fstream class. This is done as follows:

fstream finout;
finout.open(“file_name”,”opening_mode”);

OPENING AND CLOSING FILES

Opening Files using open():

File opening modes can be on of the following:

ios: : in (input)	This mode opens a file for reading. (Default for istream). The file open will be unsuccessful if we try to open a non-existing file.
ios: : out (output)	This mode opens a file for writing. (default for ofstream) . When a file is opened in this mode, it also opens in the ios::trunc mode by default. If specified already exists, it is truncated to zero length otherwise a new file will be created.
ios::app (append)	When a file is opened in this mode, the file is opened in the write mode with the file access pointer at the end of the file. This mode can be used only with output files.
ios: : binary	When a file is opened in this mode, the file is opened as a binary file and not as an ASCII text file.
ios: : ate (at the end)	When a file is opened in this mode, a file access pointer is set at the end of the file. The ios::ate is usually compiled with ios::in or ios::out for reading and writing.
ios: : trunc (truncate)	**When a file is opened in this mode, the file is truncated if a file with the specified already exists.**

READ/WRITE FROM FILE

1. Insertion operator(<<) and Extraction Operator (>>):

Like console input and output, insertion and extraction operators can also be used for the input and output operation in a file.

ofstream fout("database");

It creates fout as an object of the class ofstream and binds the fout object with the file named"database".

string str = "hello"; fout<<str; // writes str to the file database attached to fout.

2. Sequential input and output operations:

a) put(): The function put() writes a single character to the associated output file stream.

READ/WRITE FROM FILE

```
#include<iostream>
#include<fstream>
#include<string.h>
using namespace std;
int main()
{
 char text[] = “A test program for put() function.”;
 ofstream fout(“hello”); // create and open a file named hello for write operation
for( int i = 0 ; i< strlen(text) ; i++) //loop for each character
{
fout.put(text[i]); // writing each character to file using put() function;
}
}
```

b) get(): The function get() reads a single character from the associated input file stream.

READ/WRITE FROM FILE

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   ifstream infile("hello");
   char ch;
 while(1)
   {
       ch = infile.get();
       if(infile.eof() == 1)
       {
           break;
    }
    cout<<ch;

   }
   return 0;
}
```

READ/WRITE FROM FILE

3. write() and read() functions

Machines use binary format to store the information rather than ASCII format. In many cases, binary format saves disk space and makes storing and retrieval faster. To store and retrieve binary data, member functions write() and read() of ifstream and ofstream are used respectively.

write(): The write() member function is used to binary the information in a binary file. The write() is used as follows:

```
file_obj.write((char *)&variable , sizeof(variable));
```

write() function takes two arguments:

- The first is the address of the variable (The address of variable must be cast to type (char *) i.e. pointer to character type.
- The Second is the size of the variable

READ/WRITE FROM FILE

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 ofstream file;
char name[20];
int age;  float
salary;

 //opening a binary file in output mode
file.open("test.bin", ios::binary);
```

```
 //reading input from the user
cout<<"Enter name: ";
cin>>name;
cout<<"Enter age: ";
cin>>age;
cout<<"Enter salary: ";
cin>>salary;

 //writing the information to the binary file using write()
 file.write(name, sizeof(name));
 file.write((char*)(&age), sizeof(age));
file.write((char*)(&salary), sizeof(salary));

 //closing the file
file.close();
return 0;
}
```

READ/WRITE FROM FILE

In the above program, the statement

file.open("test.bin", ios::binary);

opens binary file “test.bin” for writing in the binary mode

And the statements,

file.write(name, sizeof(name));

file.write((char*)(&age), sizeof(age));

file.write((char*)(&salary), sizeof(salary));

writes the values of variables name, age and salary to the disk file “test.bin”. The first argument write() function is the pointer to the character which must take the address of the character variable. Hence, in this program, the address of the integer variable age and float variable salary must be casted to type char*.

read(): The read() member function is used to binary the information in a binary file. The read() is used as follows:

file_obj.read((char *)&variable , sizeof(variable));

READ/WRITE FROM FILE

Likewise write() function, read() function also takes two arguments:

- The first is the address of the variable (The address of variable must be cast to type (char *) i.e. pointer to character type.
- The Second is the size of the variable

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 fstream file;
char name[20] = "Jack";
int age = 35;
float marks[4] = {23,78,56,34};

char name1[20];
int age1;
float marks1[5];
```

```
//opening a binary file in output mode
file.open("test.bin", ios::binary|ios::out);

 //writing information to binary file using write()
file.write(name, sizeof(name));

 file.write((char*)(&age), sizeof(age));
 file.write((char*)(marks), sizeof(marks));

 //closing the file
 file.close();

 //opening a binary file in input mode
 file.open("test.bin", ios::binary|ios::in);
```

```
//reading the information from the binary
file using read()
file.read(name1, sizeof(name1));
file.read((char*)(&age1), sizeof(age1));
file.read((char*)(marks1), sizeof(marks1));

 cout<<"Name:"<<name1<<endl;
 cout<<"Age:"<<age1<<endl;
 cout<<"Marks:"<<endl;
   for(int i=0;i<4;i++)
   {
      cout<<marks1[i]<<endl;
 }
 file.close();
 return 0;
}
```

FILE ACCESS POINTERS AND THEIR MANIPULATORS

Each file has two associated pointers known as file pointers. One of them is called input pointer(or get pointer) and the other is called the output pointer(or put pointer). The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. When input and output operation takes place, the appropriate pointer is automatically set according to mode. For example, when we open a file in reading mode get file pointer is automatically set to the start of file. And when we open in append mode the put file pointer is automatically set at the end of file.

In C++ there are some manipulators by which we can control the movement of pointer.
The available manipulators in C++ are:

1. seekg( ): Moves get pointer(input) to a specified location.
2. seekp( ): Moves put pointer(output) to a specified location.
3. tellg( ): Gives the current position of the get pointer.
4. tellp( ): Gives the current position of the put pointer.

FILE ACCESS POINTERS AND THEIR MANIPULATORS

For Example,

```
ifstream infile;
infile.open(“test.txt”);
infile.seekg(10);
```

Moves the input file pointer to the byte number 10. Remember, the bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing to the 11th byte in the file.

```
#include <iostream>
#include <conio.h>

#include <fstream>
using namespace std;

int main( )
{

    ofstream fout(“Test.txt”) ;
    fout<< “I am Ram” ;

    fout.seekp(3) ;

    fout<< “is” ;
    int p=fout.tellp( ) ;

    cout<<p ;
    fout.close( ) ;

    getch( ) ;
}
```

Output:

5

FILE ACCESS POINTERS AND THEIR MANIPULATORS

seekp( ) and seekg( ) can also be used with two arguments as follows: *seekg (offset, refposition) ; seekp (offset, refposition) ;* Here, the parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter **refpoisition**. The refposition takes one of the following three constants defined in the ios class:

ios::beg start of the file
ios:: cur current position of the pointer
ios::end end of the file

NOTE: If we don't provide the value of refposition, then the default value will be ios::beg

The offset can also be negative as follows:
file.seekg (-5, ios: : cur) ;
This statement means the file pointer moves five bytes back from the current position.
file.seekg(0, ios: : beg) ; go to start
fout.seekg(0, ios: : cur) ; stay at current position
fout.seekg(0, ios: : end) ; go to end

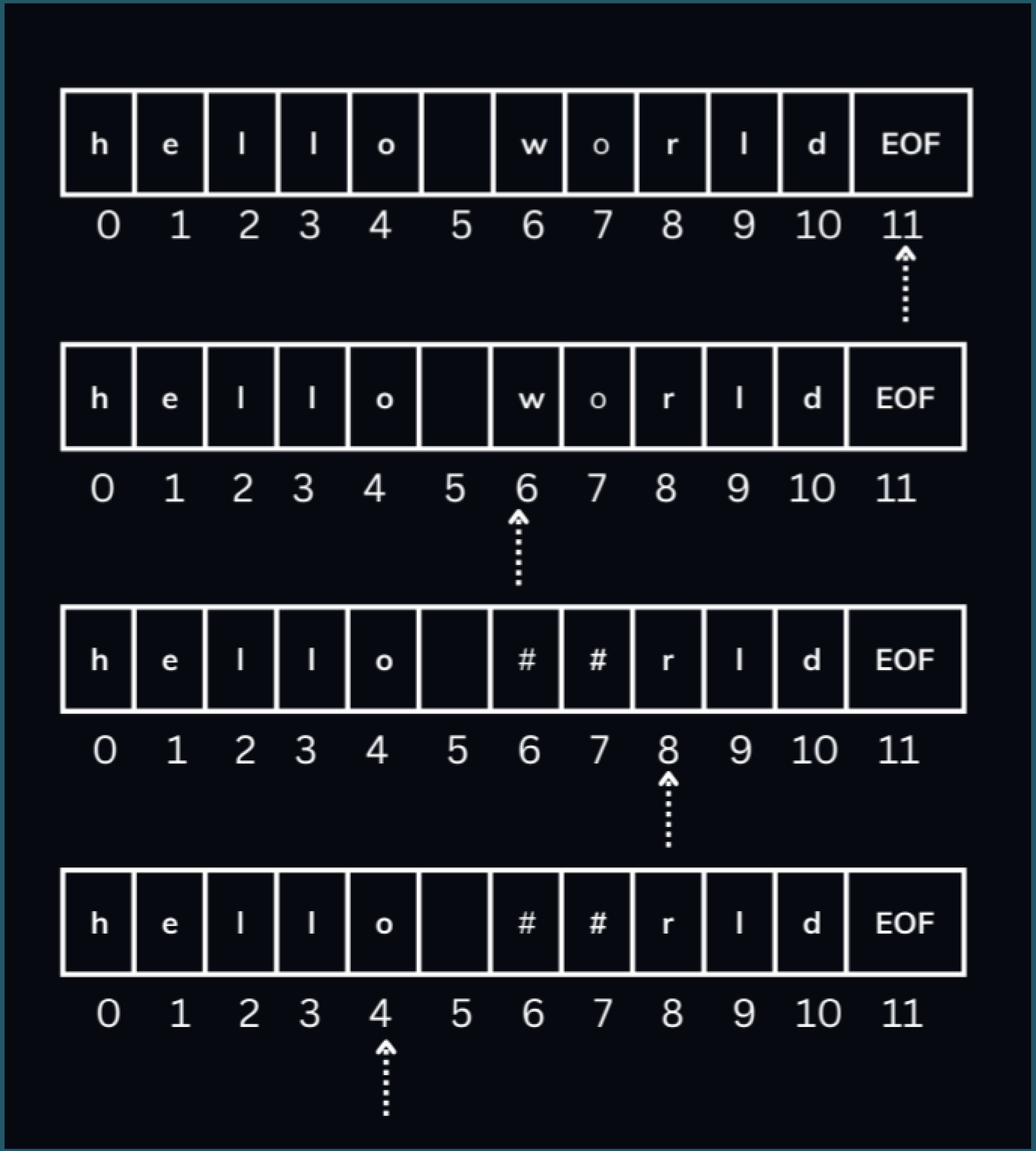
Similarly, all above concepts are same for seekp( ) ;

FILE ACCESS POINTERS AND THEIR MANIPULATORS

FOR EXAMPLE:

```
fstream fout(“abc.txt”,ios::in|ios::out);
fout<<”hello world”;
int a = fout.tellp();
cout<<a<<endl;
fout.seekp(6);
fout<<”##”;
fout.seekp(-4,ios::cur);
char ch = fout.get();
cout<<ch;
```

Output:
11
o



TESTING ERRORS DURING FILE OPERATION

So far we have been opening and using the files for reading and writing on assumption that everything is fine with the files. This may not always be true. There are many situations where errors may occur during file operations. These errors must be detected and appropriate action must be taken to prevent it. Some functions that can be used to detect errors are:

1. is_open():

Returns non-zero value if the file is opened successfully else returns zero value.

```
ofstream outfile("hello.txt");
if(outfile.is_open()) {

   //file is opened successfully
}
else
{
    //file cannot be opened..
}
```

TESTING ERRORS DURING FILE OPERATION

2. eof():

Returns non-zero if end of file is reached otherwise returns zero value.

```
ifstream fint("abc.txt");
while(! fin.eof())
{
 char ch = fin.get();
 cout<<ch;
}
else
{
  //end of file is reached
}
```

The above program reads individual characters from the file and prints on the screen until the end of file is reached.

TESTING ERRORS DURING FILE OPERATION

3. fail(): Returns true if input or output operation has failed.

```
ofstream o_file(“hello.txt”) ;
 o_file<<”I am writing on the file”;
if( o_file.fail())
{
cerr<<”can’t write on the file”;       //cerr is used to print error message on console(monitor)
}
```

4. bad(): Returns true if an invalid operation is attempted or any unrecoverable error has occurred.

5. good(): Returns true if no error has occurred. If file.good() returns true, then everything is fine and we can perform input and output operations.

Example

```
ifstream infile;
infile.open("ABC");
while(!infile.fail())
{
 //process the file
}
if(infile.eof())
{
 //terminate program normally
}
else if(infile.bad())
{
 //report fatal error
}
else
{
 infile.clear(); //clear error state so further operations can be attempted
}
```

Multiple Choice Questions on File Handling in C++

1. Which operator is used to insert the data into file?
a) >>
b) <<
c) <
d) >

2. Which function is used to position back from the end of file object?
a) seekg
b) seekp
c) both seekg & seekp
d) Seekf

3. How many objects are used for input and output to a string?
a) 1
b) 2
c) 3
d) 4

4. Which header file is used for reading and writing to a file?
a) #include<iostream>
b) #include<fstream>
c) #include<file>
d) #include<fe>

5. What will be the output of the following C++ code?

```
#include<iostream>
#include <fstream>
using namespace std;
int main ()    {
    ofstream outfile ("test.txt");
    for (int n = 0; n < 100; n++)      {
        outfile << n;
        outfile.flush();
    }
    cout << "Done";
    outfile.close();
    return 0;
}
```

a) Done
b) Error
c) Runtime error
d) DoneDoneDone

6.By seeing which operator thus this C++ program stops getting the input?

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()   {
    char ch;
    streambuf * p;
    ofstream os ("test.txt");
    pbuf = os.rdbuf();
    do {
        ch = cin.get();
        p -> sputc(ch);
    } while (ch != '.');
    os.close();
    return 0;
}
```

a) dot operator
b) insertion operator
c) $ symbol
d) @ symbol

7. What is meant by ofstream in c++?
a) Writes to a file
b) Reads from a file
c) Writes to a file & Reads from a file
d) delete a file

8. What must be specified when we construct an object of class ostream?
a) stream
b) streambuf
c) memory
d) steamostream

9. How many types of output stream classes are there in c++?

a) 1
b) 2
c) 3
d) 4

10. What is the output of this C++ program in the “test.txt” file?

```
#include <fstream>
using namespace std;
int main ()   {
    long pos;
    ofstream outfile;
    outfile.open ("test.txt");
    outfile.write ("This is an apple",16);
    pos = outfile.tellp();
    outfile.seekp (pos - 7);
    outfile.write (" sam", 4);
    outfile.close();
    return 0;
}
```

a) This is an apple
b) Apple
c) sample
d) This is a sample

11. Which is used to get the input during runtime?

a) cout
b) cin
c) coi
d) cinout

12. Where does a cin stops it extraction of data?

a) By seeing a blank space
b) By seeing (
c) By seeing a blank space & (
d) By seeing <

13. Which operator is used for input stream?

a) >
b) >>
c) <
d) <<

14. What can be used to input a string with blankspace?

a) inline
b) getline
c) putline
d) setline

15. What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main( )    {
    char line[100];
    cin.getline( line, 100, 't' );
    cout << line;
    return 0;
}
```

a) 100
b) t
c) It will print what we enter till character t is encountered in the input data
d) 200

16. Which of the following is used to create an output stream?
a) ofstream
b) ifstream
c) iostream
d) fsstream

17. Which of the following is used to create a stream that performs both input and output operations?
a) ofstream
b) ifstream
c) iostream
d) fstream

18. By default, all the files in C++ are opened in ________ mode.
a) Text
b) Binary
c) ISCII
d) VTC

19. What is the return type open() method?

a) int
b) char
c) bool
d) float

20. Which of the following is not used to seek file pointer?
a) ios::set
b) ios::end
c) ios::cur
d) ios::beg

21. Which of the following is the default mode of the opening using the ifstream class?

a) ios::in
b) ios::out
c) ios::app
d) ios::trunc

22. Which function is used in C++ to get the current position of file pointer in a file?

a) tell_p()
b) get_pos()
c) get_p()
d) tell_pos()

23. Which function is used to reposition the file pointer?

a) moveg()
b) seekg()
c) changep()
d) go_p()

24. Which of the following is used to move the file pointer to start of a file?

a) ios::beg
b) ios::start
c) ios::cur
d) ios::first

25. Which function allows you to set minimum width for the next input?

a) setfill
b) setw
c) setwidth
d) setheight

26. Which of the following is used to left-justify the output field in C++?

a) ios::scientific
b) ios::right
c) ios::left
d) none

THANK YOU