MODULE 4

I/O Streams

SYLLABUS:

C++ Class Hierarchy, File Stream, Text File Handling, Binary File Handling during file operations, Pointer types-uses, Dynamic memory allocation techniques, Garbage collection, Linked list, Generic pointers

1.1 Introduction

In C++, I/O (Input/Output) Streams are used to perform input and output operations, such as reading from the keyboard or writing to the screen. The term stream refers to a flow of data — either coming in (input) or going out (output).

iostream stands for standard input-output stream. #include iostream declares objects that control reading from and writing to the standard streams. In other words, the iostream library is an object-oriented library that provides input and output functionality using streams. A stream is a sequence of bytes

1.2 C++ Class Hierarchy

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file iostream.h. Figure given below shows the hierarchy of these classes.

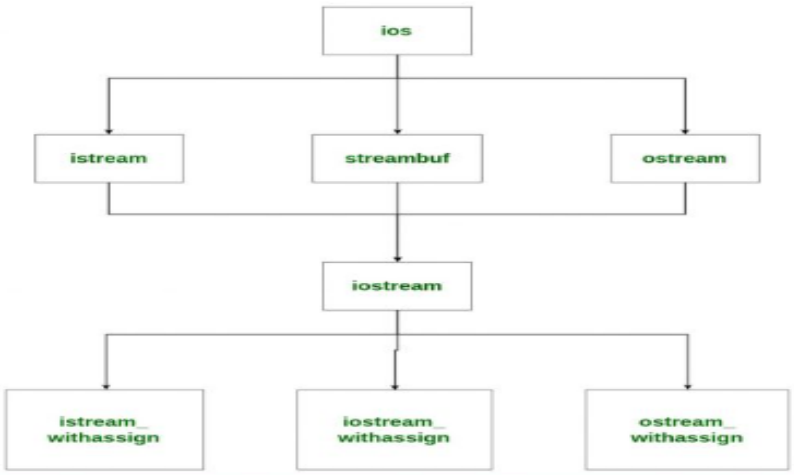


Fig : Hierarchy of iostream.h classes

ios class is topmost class in the stream classes hierarchy. It is the base class for **istream, ostream,** and **streambuf** class. **istream** and **ostream** serves the base classes for **iostream** class. The class **istream** is used for input and **ostream** for the output. Class **ios** is indirectly inherited to **iostream** class using **istream** and **ostream**. To avoid the duplicity of data and member functions of **ios** class, it is declared as virtual base class when inheriting in **istream** and **ostream** as follows :

```
class istream: virtual public ios
{
};
class ostream: virtual public ios
{
};
```

The **_withassign classes** are provided with extra functionality for the assignment operations that's why **_withassign** classes.

Facilities provided by these stream classes

The ios class: The ios class is responsible for providing all input and output facilities to all other stream classes.

The istream class: This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as **get, getline, read, ignore, putback** etc..

Example :

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

int main()
{
    char x;

    // used to scan a single char
    cin.get(x);

    cout << x;
}
```

The ostream class: This class is responsible for handling output stream. It provides number of function for handling chars, strings and objects such as **write, put** etc..

Example :

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

int main()
{
    char x;

    // used to scan a single char
    cin.get(x);

    // used to put a single char onto the screen.
    cout.put(x);
}
```

The iostream: This class is responsible for handling both input and output stream as both **istream class** and o**stream class** is inherited into it. It provides function of both **istream class** and o**stream class** for handling chars, strings and objects such as **get, getline, read, ignore, putback, put, write** etc..

Example:

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

int main()
{

// this function display
// ncount character from array
cout.write("Hello", 5);
}
```

istream_withassign class: This class is variant of **istream** that allows object assignment. The predefined object **cin** is an object of this class and thus may be reassigned at run time to a different **istream** object.

Example: To show that **cin** is object of **istream** class

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

class demo {
public:
    int dx, dy;

    // operator overloading using friend function
    friend void operator>>(demo& d, istream& mycin)
    {
        // cin assigned to another object mycin
        mycin >> d.dx >> d.dy;
    }
};
int main()
{
    demo d;
    cout << "Enter two numbers dx and dy\n";

    // calls operator >> function and
    // pass d and cin as reference
    d >> cin; // can also be written as operator >> (d, cin) ;

    cout << "dx = " << d.dx << "\tdy = " << d.dy;
}
```

ostream_withassign class: This class is variant of **ostream** that allows object assignment. The predefined objects **cout, cerr, clog** are objects of this class and thus may be reassigned at run time to a different **ostream** object.

Example: To show that **cout** is object of **ostream** class.

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

class demo {
public:
    int dx, dy;

    demo()
    {
        dx = 4;
        dy = 5;
    }

    // operator overloading using friend function
    friend void operator<<(demo& d, ostream& mycout)
    {
        // cout assigned to another object mycout
        mycout << "Value of dx and dy are \n";
        mycout << d.dx << " " << d.dy;
    }
};
int main()
{
    demo d; // default constructor is called

    // calls operator << function and
    // pass d and cout as reference
    d << cout; // can also be written as operator << (d, cout) ;
}
```

1.3 File Stream

Many real-life problems handle large volumes of data and, in such situations, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept offiles. 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.

The I/O system of C++ handles file operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream. the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file. This is illustrated in the following Fig.

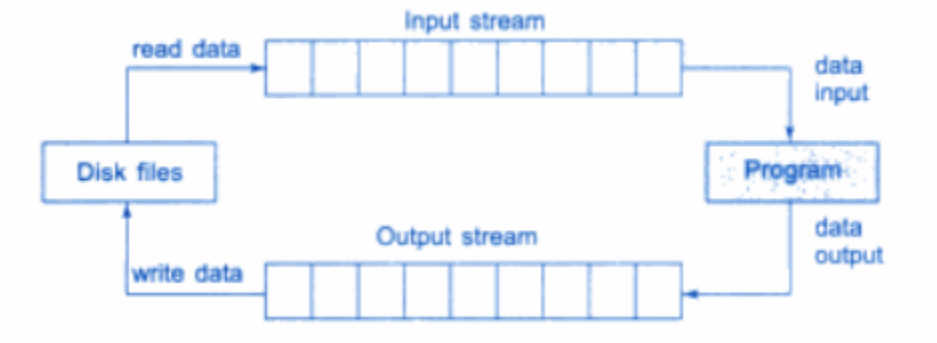


Fig : File Input and Output Streams

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

Classes for File Stream Operations

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. 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.

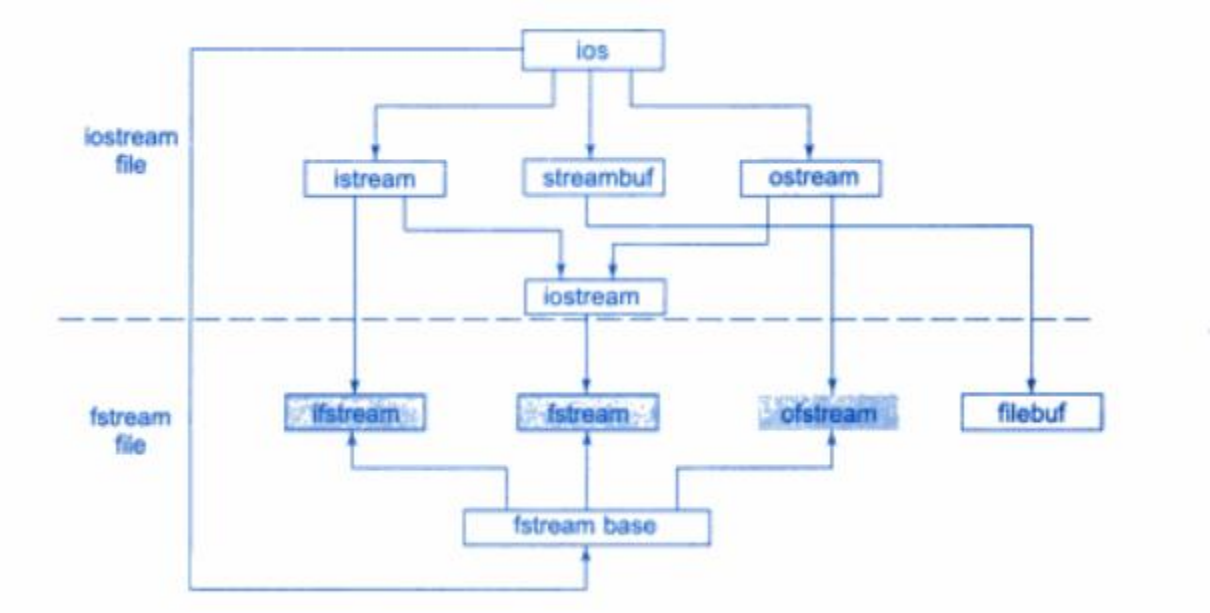


Fig : Stream classes for file operations (contained in fstream file)

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to the file streams. Serves as a base for **fstream, ifstream** and **ofstream** class. Contains **open()** and **close()** functions.
ifstream	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(), getline(), read(), seekg()** and **tellg()** functions from **istream**.
Ofstream	Provides output operations. Contains **open()** with default output mode. Inherits **put(), seekp(), tellp(),** and **write(),** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations. Contains **open()** with default input mode. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

Fig : Details of file stream classes

Opening and Closing a File

If we want to use a disk file, we need to decide the following things about the file and its intended use:

1. Suitable name for the file

2. Data type and structure

3. Purpose

4. Opening method

The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension.

Example :

Input.data

Test.doc

Student.cpp

salary

Output

For opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes ifstream, ofstream, and fstream that are contained in the header file fstream. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

1. Using the constructor function of the class.

2. Using the member function open() of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

Opening Files Using Constructor

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. That is to say, the class ofstream is used to create the output stream and the class ifstream to create the input stream.

2. Initialize the file object with the desired filename.

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

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

This creates outfile as an ofstream object that manages the output stream. This statement also opens the file results and attaches it to the output stream outfile.

Similarly, the following statement declares infile as an ifstream object and attaches it to the file data for reading (input).

```
ifstream infile("data"); // input only
```

The program may contain statements like:

```
outfile << "TOTAL";

outfile << sum;

infile >> number;

infile >> string;
```

The connection with a file is closed automatically when the stream object expires (when the program terminates).

```
// Creating files with constructor function

#include <iostream.h>
#include <fstream.h>

int main()
{
        ofstream outf("ITEM");          // connect ITEM file to outf

        cout << "Enter item name:";
        char name[30];
        cin >> name;                    // get name from key board and

        outf << name << "\n";           // write to file ITEM

        cout << "Enter item cost:";
        float cost;
        cin >> cost;                    // get cost from key board and

        outf << cost << "\n";           // write to file ITEM

        outf.close();                   // Disconnect ITEM file from outf

        ifstream inf("ITEM");           // connect ITEM file to inf

        inf >>  name;                   // read name from file ITEM
        inf >> cost;                    // read cost from file ITEM
         cout << "\n";
         cout << "Item name:" << name << "\n";
         cout << "Item cost:" << cost << "\n";

         inf.close();                        // Disconnect ITEM from inf

         return 0;
}
```

Output :

Enter item name: CD-ROM

Enter item cost:250

Item name: CD-ROM

Item cost:250

When a file is opened for writing only, a new file is created if there is no file ofthat name. Ifa file by that name exists already, then its contents are deleted and the file is presented as a clean file.

Opening Files Using open()

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 files sequentially. In such cases, we may create a single stream object and use it to open each file in turn.

```
// Creating files with open() function

#include <iostream.h>
#include <fstream.h>

int main()
{
  ofstream fout;                        // create output stream
  fout.open("country");                 // connect "country" to it
  fout << "United States of America\n";
  fout << "United Kingdom\n";
  fout << "South Korea\n";

  fout.close();                         // disconnect "country" and

  fout.open("capital");                 // connect "capital"

  fout  << "Washington\n";
  fout  << "London\n";
  fout  << "Seoul\n";

  fout.close();                         // disconnect "capital"

  // Reading the files
  const int N = 80;                     // size of line
  char line[N];
```

```
ifstream fin;                          // create input stream
fin.open("country");                   // connect "country" to it

cout <<"contents of country file\n";

while(fin)                             // check end-of-file
{
      fin.getline(line, N);            // read a line
      cout << line ;                   // display it
}

fin.close();                           // disconnect "country" and
```

```
   fin.open("capital");                // connect "capital"

   cout << "\nContents of capital file \n";

   while(fin)
   {
         fin.getline(line, N);
         cout << line ;
   }
   fin.close();

   return 0;
}
```

Output :

Contents of country file

United States of America

United Kingdom

South Korea

Contents of capital file

Washington

London

Seoul

File Modes

The general form of the function open() with two arguments is:

stream-object.open("filename", mode);

The second argument mode (called file mode parameter) specifies the purpose for which the file is opened. The prototype of class member functions contain default values for the second argument and therefore it uses the default values in the absence of the actual values.

The default values are as follows:

ios::in for ifstream functions meaning open for reading only.

ios::out for ofstream functions meaning open for writing only.

The file mode parameter can take one (or more) of such constants defined in the class ios.

Parameter	Meaning
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

Fig : File mode parameters

1.4 Text File Handling

Text File Handling is a process in which we create a text file and store data permanently on a hard disk so that it can be retrieved from the memory later for use in a program. In a text file, whatever data we store is treated as text. Even if we store numbers, it is treated as text.

Write to a file

```
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

int main()
{
    fstream fs;
    char str[200];

    fs.open("e:/data.txt", ios::app);

    if(fs.is_open()==0)
    {
        cout<<"Cannot open file";
    }
    else
    {
        cout<<"Enter a few lines of text:\n";
        while(strlen(gets(str)) > 0)
        {
            fs<<str<<endl;
        }
        fs.close();
    }
    return 0;
}
```

Read from a file

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream fs;
    char c;

    fs.open("e:/data.txt", ios::in);

    if(fs.is_open()==0)
    {
        cout<<"Cannot open file";
    }
    else
    {
        while(!fs.eof())
        {
            fs.get(c);
            cout<<c;
        }
        fs.close();
    }
    return 0;
}
```

Rename a file

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

using namespace std;

int main()
{
    rename("E:/data.txt", "E:/data2.txt");
    printf("File renamed successfully");
    return 0;
}
```

Delete a file

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

using namespace std;

int main()
{
    remove("E:/data2.txt");
    printf("File Deleted Successfully");
    return 0;
}
```

1.5 Binary File handling during file operations

Binary File Handling is a process in which we create a file and store data in its original format. It means that if we stored an integer value in a binary file, the value will be treated as an integer rather than text.

Write to a file and Read from a file

To write a binary file in C++ use write() method. It is used to write a given number of bytes on the given stream, starting at the position of the "*put*" pointer. The file is extended if the put pointer is currently at the end of the file. If this pointer points into the middle of the file, characters in the file are overwritten with the new data. If any error has occurred during writing in the file, the stream is placed in an error state.

To read a binary file in C++ use read() method. It extracts a given number of bytes from the given stream and place them into the memory, pointed to by the first parameter. If any error is occurred during reading in the file, the stream is placed in an error state, all future read operation will be failed then.

gcount() can be used to count the number of characters has already read. Then clear() can be used to reset the stream to a usable state.

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

struct Student {
  int roll_no;
  string name;
};

int main() {
  ofstream wf("student.dat", ios::out | ios::binary);

  if (!wf) {
    cout << "Cannot open file!" << endl;
    return 1;
  }
```

```
Student wstu[3];

wstu[0].roll_no = 1;
wstu[0].name = "Ram";
wstu[1].roll_no = 2;
wstu[1].name = "Shyam";
wstu[2].roll_no = 3;
wstu[2].name = "Madhu";

for (int i = 0; i < 3; i++)
  wf.write((char * ) & wstu[i], sizeof(Student));

wf.close();

if (!wf.good()) {
  cout << "Error occurred at writing time!" << endl;
  return 1;
}
ifstream rf("student.dat", ios::out | ios::binary);
if (!rf) {
  cout << "Cannot open file!" << endl;
  return 1;
}

Student rstu[3];

for (int i = 0; i < 3; i++)
  rf.read((char * ) & rstu[i], sizeof(Student));
rf.close();
if (!rf.good()) {
  cout << "Error occurred at reading time!" << endl;
  return 1;
}

cout << "Student's Details:" << endl;
for (int i = 0; i < 3; i++) {
  cout << "Roll No: " << wstu[i].roll_no << endl;
  cout << "Name: " << wstu[i].name << endl;
  cout << endl;
}
```

1.6 Pointer types

Each file has two associated pointers known as the file pointers. One of them is called the input pointer (or get pointer) and the other is called the output pointer (or put pointer). We can use these pointers to move through the files while reading or writing. 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. Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start. Similarly, when we open a file in write-only mode, the existing contents are deleted and the output pointer is set at the beginning. This enables us to write to the file from the start. In case, we want to open an existing file to add more data, the file is opened in 'append' mode. This moves the output pointer to the end of the file (i.e. the end of the existing contents).

Functions for Manipulation of File Pointers

The file stream classes support the following functions to manipulate file pointers:

seekg() : Moves get pointer (input) to a specified location.

seekp() : Moves put pointer(output) to a specified location.

tellg() : Gives the current position of the get pointer.

tellp() : Gives the current position of the put pointer.

seekg() :

seekg() is a function in the iostream library that allows you to seek an arbitrary position in a file. It is included in the **<fstream>** header file and is defined for **istream class.** It is used in file handling to sets the position of the next character to be extracted from the input stream from a given file.

Syntax : istream&seekg(streampos position);

Or

istream&seekg(streamoff offset, ios_base::seekdir dir);

where,

- **position:** is the new position in the stream buffer.
- **offset:** is an integer value of type streamoff representing the offset in the stream's buffer. It is relative to the dir parameter.
- **dir:** It is the seeking direction. It is an object of type ios_base::seekdir that can take any of the following constant values.

seekp() :

The **seekp(pos)** method of **ostream in C++** is used to set the position of the pointer in the output sequence with the specified position. This method takes the new position to be set and returns this ostream instance with the position set to the specified new position.

Syntax:

ostream& seekp(streampos pos);

Parameter: This method takes the **new position** to be set as the parameter.
Return Value: This method returns **this ostream instance** with the position set to the specified new position.
Exceptions: If the operation sets an internal state flag (except eofbit) that was registered with member exceptions, the function throws an exception of member type **failure**.

tellg() :

The **tellg()** function is used with input streams, and returns the current "get" position of the pointer in the stream. It has no parameters and returns a value of the member type

pos_type, which is an integer data type representing the current position of the get stream pointer.

Syntax:-

```
pos_type tellg();
```

Returns: The current position of the get pointer on success, pos_type(-1) on failure.

tellp() :

The **tellp()** function is used with output streams, and returns the current "put" position of the pointer in the stream. It has no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the put stream pointer.

Syntax:

```
pos_type tellp();
```

Return : Current output position indicator on success otherwise return -1.

1.7 Dynamic memory allocation techniques

The means by which data object can be created as they are needed during the program execution. Such data objects remain in existence until they are explicitly destroyed. In C++, dynamic memory allocation is accomplished with the operators new (for creating data objects) and delete (for destroying them).

C++ allows us to allocate the memory dynamically in run time. This is known as dynamic memory allocation. In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

We can allocate memory dynamically using the new operator and deallocate memory dynamically using the delete operator.

new Operator

The "new" operator in C++ is used to allocate memory dynamically for a variable or an object at runtime. This means that the memory is allocated during the execution of the program, as opposed to being allocated at compile time. When the "new" operator is called, it reserves a block of memory that is large enough to hold the object being created and then returns a pointer to the first byte of that memory block.

Syntax

1) Syntax to initialize the memory

Pointer_name=new datatype;

Example

```
int *ptr=new int;
```

2) syntax to allocate a block of memory

pointer_variable = new datatype(value);

Example

```
int *ptr=new int(10);
```

Delete Operator

The delete operator is used to deallocate memory that was previously allocated on the heap using new. It takes a pointer to the memory to be deallocated as an argument.

For **example:**

```
delete p; // Deallocates the memory pointed to by p
```

The "delete" operator is used to deallocate memory that the "new" operator previously allocated. Once a block of memory has been allocated by "new," it is important to deallocate it when it is no longer needed so that other parts of the program can reuse the memory. The

"delete" operator releases the memory back to the system, and other parts of the program can use it.

Memory leaks occur when the program allocates memory dynamically but does not deallocate it properly. This causes the program to consume more memory gradually, eventually leading to poor performance or even crashing the program. Use delete to deallocate memory allocated with new to avoid memory leaks.

Example 1:

```
#include <iostream>
int main() {
    int* pInt = new int;   // dynamically allocate memory for an int
    *pnt = 5;              // store the value 5 in the allocated memory
    std::cout << *pnt;     // output the value stored in the allocated memory
    delete pnt;            // deallocate the memory to prevent memory leak
    return 0;
}
```

Output: 5

Example 2:

```cpp
#include <iostream>
int main()
{
    int* ptr1 = new int;   // dynamically allocate memory for an int
    *ptr1 = 5;  // store the value 5 in the allocated memory
   float *ptr2 = new float(20.324);
   int *ptr3 = new int[28];
   std::cout << "Value of pointer variable 1 : " << *ptr1<<std::endl;
   std::cout << "Value of pointer variable 2 : " << *ptr2<<std::endl;
   if (!ptr3)
   std::cout << "Allocation of memory failed\n";
   else {
      for (int i = 1; i < 15; i++)
      ptr3[i] = i+1;
      std::cout << "Value of store in block of memory: ";
      for (int i = 1; i < 15; i++)
      std::cout << ptr3[i] << " ";
   }
   std::cout << *ptr1;     // output the value stored in the allocated memory
   delete ptr1;// deallocate the memory to prevent memory leak
   delete ptr2;
   delete ptr3;
   return 0;
}
```

Output

```
Value of pointer variable 1 :
5
Value of pointer variable 2 :
20.324
Value of store in block of memory:
2 3 4 5 6 7 8 9 10 11 12 13 14 15 5
```

1.8 Garbage collection

C++ does not have automatic garbage collection, meaning programmers are responsible for manually managing memory allocation and deallocation, which can lead to memory leaks if not done correctly.

Unlike languages like Java or Python, C++ doesn't have a built-in garbage collector that automatically reclaims memory when it's no longer in use. C++ programmers must explicitly allocate memory using new and deallocate it using delete. If memory is allocated but not deallocated when it's no longer needed, it can lead to memory leaks, where the program continues to consume memory even after it's no longer required.

Why manage garbage collection?

Managing garbage collection in C++ is important for several reasons. First, it can help you avoid memory leaks, which occur when you allocate memory but forget to free it when it is no longer needed. Memory leaks can cause performance degradation, resource exhaustion, and unexpected errors. Second, it can help you reduce memory fragmentation, which occurs when you allocate and free memory in different sizes and patterns. Memory fragmentation can make it harder to find contiguous blocks of memory for new allocations and increase the overhead of memory management. Third, it can help you improve memory efficiency, which means using the optimal amount of memory for your program's needs and avoiding wasting memory on unused or unnecessary data.

Managing garbage collection manually

Managing garbage collection in C++ manually, using the operators **new** and **delete** , provides full control over memory allocation and deallocation. However, it can be a tedious and time-consuming task, particularly for complex programs that involve multiple layers of abstraction, inheritance, polymorphism, and dynamic allocation. Common mistakes that can lead to memory leaks or corruption include forgetting to delete a pointer or an object after use, deleting a pointer or an object more than once, deleting a pointer or an object

that was not allocated with **new** , deleting a pointer to a base class without a virtual destructor, deleting a pointer to an array without using **delete[]** , using dangling pointers that point to freed memory, or using uninitialized or invalid pointers.

Managing garbage collection with smart pointers

In C++, another way to manage garbage collection is through smart pointers. These objects wrap around raw pointers and automatically free the memory they point to when they go out of scope or are reassigned. Smart pointers can simplify memory management and reduce the risk of memory leaks or corruption, but they also come with some drawbacks, such as overhead to memory operations, creating circular references that prevent memory from being freed, introducing compatibility issues with legacy code or libraries that use raw pointers, and changing the semantics and behavior of pointer operations. C++ provides several types of smart pointers with different characteristics and use cases. The most common ones are **std::unique_ptr** , which is useful for managing resources that have a single owner and a clear lifetime; **std::shared_ptr** , which is useful for managing resources that have multiple owners and a dynamic lifetime; and **std::weak_ptr** , which does not own the memory it points to and does not affect the reference count of a **std::shared_ptr** . It can be used to break circular references or to access resources that may or may not exist.

Managing garbage collection with RAII

RAII stands for Resource Acquisition Is Initialization, a programming idiom that binds the lifetime of a resource to the lifetime of an object. In C++, this means using constructors to acquire resources and destructors to release them, so that resources are automatically freed when the objects that own them are destroyed. RAII can be used to manage not only memory, but also other types of resources, such as files, sockets, locks, threads, etc. This programming technique can offer several advantages, such as ensuring that resources are always released even in the case of exceptions or early returns.

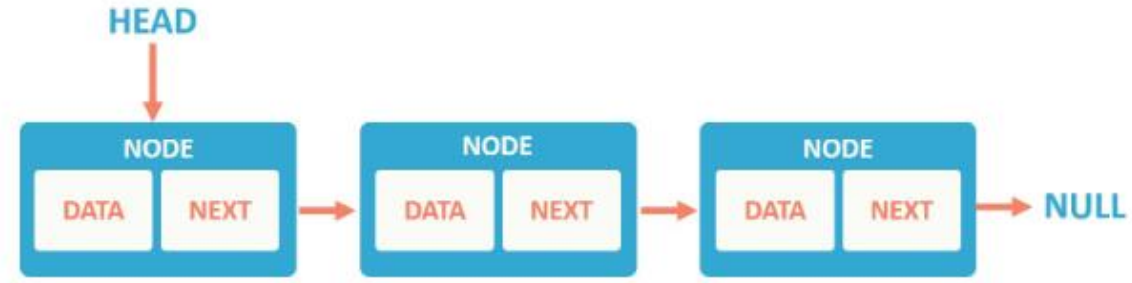
It also avoids the need to write explicit cleanup code or use **try-finally** blocks, making the code more readable, concise and consistent while supporting the principle of single responsibility and separation of concerns. To implement RAII in C++, you can either use

existing RAII classes like smart pointers, containers, streams and locks or create your own by defining a class that represents a resource and encapsulating its acquisition and release in the constructor and destructor respectively.

Additionally, you should disable or appropriately implement the copy constructor and copy assignment operator while providing accessors and operators to manipulate the resource as needed.

1.9 Linked list

Linked List is a linear data structure that contains two parts: a collection of data called **Nodes** and a **Reference pointer** that holds the address of the next node in the sequence.



Properties of Linked List

- **Dynamic size**: Linked lists are dynamic in nature, as they can expand and contract as required.
- **Non-contiguous storage**: Linked list nodes are stored in a non-contiguous storage.
- **Effective insertion and deletion**: When compared to arrays, a linked list easily allows the insertion and deletion of nodes.
- **Flexible**: Data structures, like stacks, queues, and hash tables, can be implemented using linked lists.
- **Access Time**: Unlike arrays, accessing elements of a linked list is less efficient as we need to traverse starting from the linked list HEAD.

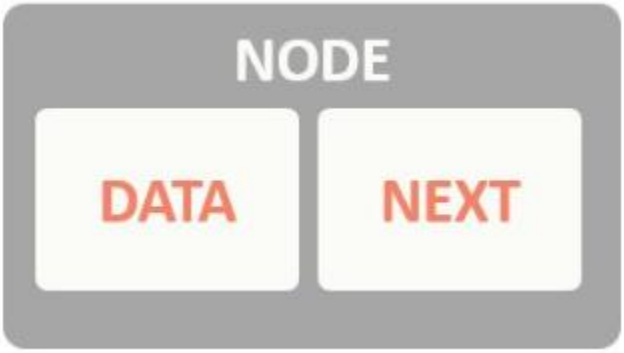
- **Memory Overhead**: As the data and the pointers of a linked list need to be stored in the case of a linked list, the memory required to store a linked list is higher than an array.

Traversal Direction: Traversal of a linked list is generally uni-directional as we have to start from the HEAD and iterate until we reach the linked list's tail.

Representation of Linked List

A linked list is a chain of nodes, where each node has the 2 below-listed components.

- **Data:** This is where the data is stored
- **Next:** This is where the address of the next node is stored.



A linked list with a single Node can be defined as follows in C++

```
struct node
{
  int data;
  struct node *next;
};
```

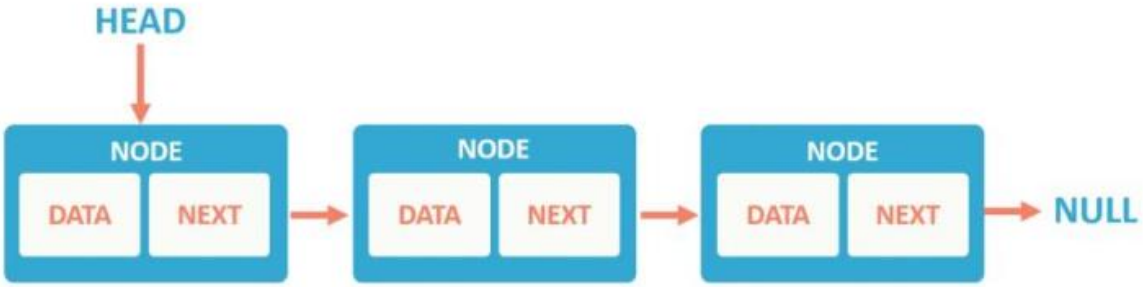
Types of Linked Lists in C++

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Singly Linked List

A singly linked list is a node-based linear data structure. Each node in the list has a data field and a pointer to the following node. The head and tail of a list are the first and last nodes, respectively.

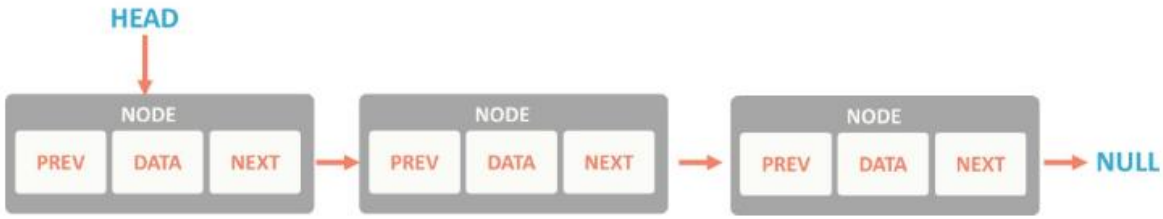
Each node in singly linked lists has just one link, which points to the node after it in the list, hence the name "singly". Singly-linked lists are now less effective for some operations, including accessing entries in the centre of the list, but they are also easier to create as a result.



Doubly Linked List

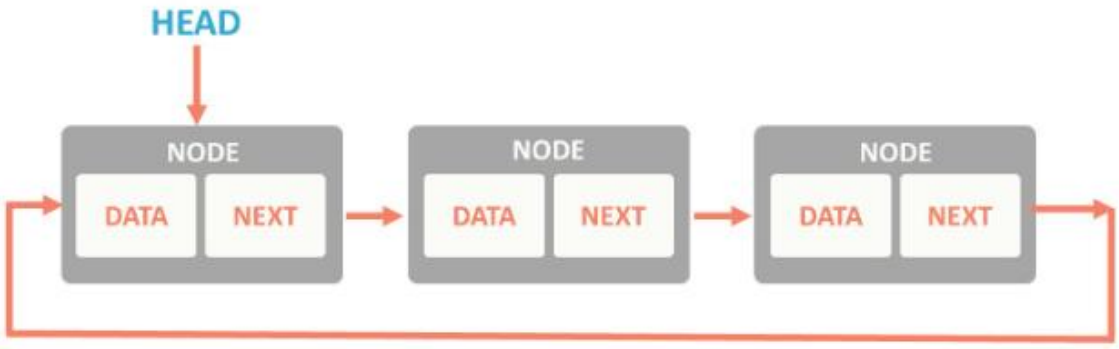
A doubly linked list is a node-based linear data structure. Every node in the list has two references—one to the node before it and the other to the node after it—and a data field. The head and tail of a list are the first and last nodes, respectively.

The term "doubly" linked lists refers to every node in the list containing two links: one to the node before it and another to the node after it. Because of this, doubly linked lists are more effective for certain actions (such as retrieving members from the centre of the list), but their implementation becomes more difficult.



Circular Linked List

A circular linked list is a variant of a linked list where the list's final node is looped back around to connect to the initial node. The last node in a conventional singly linked list usually has a null pointer (or equivalent) as its subsequent pointer to denote the list's end. A continuous loop is created in a circular linked list when the subsequent pointer from the last node points back towards the first node.



Basic Operations on a List

- Traversing a Linked List
- Inserting an item in the Linked List
- Deleting an item from the Linked List

Traversing a Linked List

To traverse a linked list in C++, we can make use of the following steps::

- Create a pointer to the head node of the linked list.
- Unless the pointer is null, continue with the following:
  - Get/print the data at the current node
  - Change the pointer to the subsequent node of the linked list
- Stop the traversal ifd the pointer points to null.

Algorithm

The algorithm typically looks like the following:

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: Apply Process to PTR DATA

Step 4: SET PTR = PTR NEXT

[END OF LOOP]

Step 5: EXIT

```
// This program traverses a linked list and prints the data field of each node.

#include  < iostream >

struct Node {
  int data; // The data field of the node.
  Node* next; // A pointer to the next node in the linked list.
};

// Traverses the linked list and prints the data field of each node.
void traverse_linked_list(Node* head) {
  // Create a pointer to the current_node node in the linked list.
  Node* current_node = head;
```

```
  // While the pointer is not null, print the data field of the current_node node and
  // move the pointer to the next node in the linked list.
  while (current_node != nullptr) {
    std::cout < < current_node- > data < < " ";
    current_node = current_node- > next;
  }

  // Print a newline character.
  std::cout < < std::endl;
}

int main() {
  // Create a head node for the linked list.
  Node* head = new Node{10};

  // Add two more nodes to the linked list.
  head- > next = new Node{20};
  head- > next- > next = new Node{30};

  // Traverse the linked list and print the data field of each node.
  traverse_linked_list(head);

  return 0;
}
```

Output

10 20 30

Inserting an item in the Linked List

There are 3 different ways to insert elements to a linked list in C++. Let's discuss each one of them in detail.

Insert at the beginning of the Linked list

To insert an element at the beginning of a linked list, follow the below steps:

- Create a new node with the new data.

- Set the next pointer of the new node to the current head node.
- Set the head node to the new node.

Algorithm

A typical implementation of this process follows the below algorithm

Step 1: [INITIALIZE] CREATE NEW_NODE

Step 2: SET NEW_NODE DATA = VALUE

Step 3: SET NEW_NODE NEXT = HEAD

Step 4: SET HEAD = NEW_NODE

Step 5: EXIT

Deleting Node(s) of a Linked List

To delete a specific node from the linked list, use the following method:

- Traverse to the node that is to be deleted.
- If the node is HEAD node, set the node to the next node in the sequence.
- Else, find the previous node to the node that is to be deleted.
- Set the next pointer of the previous node to the next node of the node to be deleted.

Algorithm

For deleting an element from a linked list, make use of the below algorithm.

Step 1: SET PTR = HEAD, SET PREV = NULL

Step 2: Calculate the value to delete (e.g., VALUE)

Step 3: Repeat Steps 4 and 5 while PTR != NULL and PTR DATA != VALUE

Step 4: SET PREV = PTR

Step 5: SET PTR = PTR NEXT

[END OF LOOP]

Step 6: If PTR is NULL (Value not found)

EXIT

Step 7: If PREV is NULL (Deleting the first node)

Step 8: SET HEAD = PTR NEXT

Step 9: DELETE PTR

EXIT

Step 10: SET PREV NEXT = PTR NEXT

Step 11: DELETE PTR

Step 12: EXIT

1.10 Generic pointers

In C++, a generic pointer, also known as a void pointer, is a special type of pointer that can point to data of any type. It is declared as void *. This allows a single pointer to hold the address of different data types without requiring explicit type conversions during assignment. However, to use a void pointer, you must cast it to the appropriate type before dereferencing.

In C++, a **generic pointer** is typically used to refer to a pointer that can point to any data type. C++ does not have a direct "generic pointer" type like some languages, but you can achieve similar functionality by using the void* type or through **template classes/functions.**

Key characteristics and uses of void pointers :

1. Type-agnostic:

Void pointers don't have a specific data type associated with them, making them suitable for situations where the type of the data being pointed to is not known or can vary

- **2. Memory allocation:**

Functions like malloc() and calloc() return void pointers, allowing you to allocate memory without specifying the type of data it will store.

3. Generic functions:

Void pointers are used in C and C++ to implement generic functions, where the function can work with different data types without knowing the exact type in advance

4. qsort():

The qsort() function in the C standard library uses void pointers for the array to be sorted and the comparison function, enabling sorting of different data types.

Example :

```cpp
#include <iostream>

int main() {
    int num = 10;
    double d = 3.14;
    char ch = 'A';

    void* ptr;

    // Store the address of num in the void pointer
    ptr = &num;

    // Cast the void pointer to int* before dereferencing and printing
    std::cout << "Value of num: " << *(int*)ptr << std::endl;

    // Store the address of d in the void pointer
    ptr = &d;

    // Cast the void pointer to double* before dereferencing and printing
    std::cout << "Value of d: " << *(double*)ptr << std::endl;

    // Store the address of ch in the void pointer
    ptr = &ch;

    // Cast the void pointer to char* before dereferencing and printing
    std::cout << "Value of ch: " << *(char*)ptr << std::endl;

    return 0;
}
```

void* (Void Pointer)

A void* is a special type of pointer in C++ that can point to any data type, but it does not have a type itself. This allows it to point to any object, but it cannot be dereferenced directly without type casting.

Example

```cpp
#include <iostream>

void printData(void* ptr, const char* type) {
    if (type == "int") {
        std::cout << "Integer: " << *(static_cast<int*>(ptr)) << std::endl;
    } else if (type == "double") {
        std::cout << "Double: " << *(static_cast<double*>(ptr)) << std::endl;
    }
}

int main() {
    int x = 10;
    double y = 3.14;

    // Creating a void pointer
    void* ptr;

    // Pointing to an int
    ptr = &x;
    printData(ptr, "int");

    // Pointing to a double
    ptr = &y;
    printData(ptr, "double");

    return 0;
}
```

Output

```
Integer: 10
Double: 3.14
```

Template Function for Generic Pointers

If you want to write a function that works with any type in a more type-safe manner, you can use **templates**. Templates allow you to write generic code that works with any data type.

Example

```cpp
#include <iostream>

// Template function to print the data of any type
template <typename T>
void printData(T* ptr) {
    std::cout << "Value: " << *ptr << std::endl;
}

int main() {
    int x = 10;
    double y = 3.14;

    // Using template function with int pointer
    printData(&x);

    // Using template function with double pointer
    printData(&y);

    return 0;
}
```

Output

```
Value: 10
Value: 3.14
```

Template Class for Generic Pointers

If you need a more complex structure or class that works with different types, you can define a **template class**.

```
#include <iostream>

template <typename T>
class GenericPointer {
private:
    T* ptr;
public:
    // Constructor
    GenericPointer(T* p) : ptr(p) {}

    // Dereference the pointer and return the value
    T getValue() {
        return *ptr;
    }

    // Destructor
    ~GenericPointer() {
        std::cout << "Pointer to " << typeid(T).name() << " is being deleted" << std::endl;
    }
};
```

```
int main() {
    int x = 10;
    double y = 3.14;

    // Create generic pointers
    GenericPointer<int> intPtr(&x);
    GenericPointer<double> doublePtr(&y);

    std::cout << "Integer Value: " << intPtr.getValue() << std::endl;
    std::cout << "Double Value: " << doublePtr.getValue() << std::endl;

    return 0;
}
```

Output

```
Integer Value: 10
Double Value: 3.14
Pointer to i is being deleted
Pointer to d is being deleted
```

Model Questions

1. Explain C++ Class Hierarchy
2. Explain with diagram File Input Streams
3. Explain with diagram File Output Streams
4. List Classes for File Stream Operations
5. Explain different file stream classes
6. Explain new operator
7. Explain delete operator

8. Explain garbage collection

9. What is dynamic memory management?

10. Explain advantages of pointers.

11. What are the Key characteristics of void pointers?

12. Write an algorithm, to delete a node from linked list.

13. Write an algorithm, to inserting an item to a linked list.

14. Write an algorithm, to traverse a linked list.

15. Explain circular linked list.