CPP101

Coding Club, IIT Guwahati

Coding Club
IIT Guwahati

```cpp
#include <iostream>
#include <functional>

int main()
{
    int C;
    std :: cin >> C;

    std::function<void(int)> Advanced = [&](int C) {
        C++;
    };

    Advanced(C++);

    return 0;
}
```

Week 3

Day 1

Lambda Expression in C++

C++ 11 introduced lambda expressions to allow inline functions which can be used for short snippets of code that are not going to be reused and therefore do not require a name. These are very helpful when you have to define a function inside a function itself and also need all the local variables of the current function

- Generally, the return type of lambda functions is determined by the compiler itself but in case of multiple return statements, the return type needs to be explicitly declared
- There are several ways to access all the variable that lie in the function that encloses the lambda function
  - [&] -> can access all the variables by reference
  - [=] -> can access all the variables by value
  - [a,&b] -> capture value of a by value and b by reference
  -

Here is a template how a lambda function is declared:

```
[Capture Clause](parameter list) mutable exception -> return_type{
  Method definiton
 }
  // Capture clause: Lambda introducer as per C++ specification.
  // Parameter list: Is optional and is similar to the parameter list of a method.
  // Mutable: Optional. Enables variables captured by a call by value to be
modified.
  // exception: Exception specification. Optional. Use "noexcept" to indicate that
lambda does not throw an exception.
  // return_type : Optional but needed in some complex functions
  // Method definition: Lambda body.
```

Learn more about lambda here
Also see this video to clear your concepts

You can learn more about capture clause here

Why lambda functions are better than normal functions?

Lambda functions are apparently better optimised by the C++ compiler because these are nothing just function objects. Learn more about it here

Also you can follow this guide to learn how lambda functions are used in competitive programming ⟶ USACO Guide

Some notes:

- Earlier you cannot use templates while defining the lambda function but from C++-20 it is possible [discussion]
- You need to be very careful when you are using a lambda function as a recursive function because compiler needs to know that there exists such function in the enclosing domain so either you pass the lambda function in the argument list itself or first declare the function and add it's body later [Read more about it here]
- Lambda vs Functors [discussion]

Move and Copy Semantics

Revisiting l-value and r-value in C++

You can recap your prior knowledge of l-value and r-value studied in week 2 by this video or this article

You can also refer to this article

Move Semantics

Whenever you assign a constant value to some object the value is copied to target object which is a costly operation. Many times there is no need to copy the rvalue to an object's data, we just need to move it's value, this is where move semantics comes in
Refer to these videos to get an introduction to move semantics video1 and video2

Now go through this article if you want to have a more clear understanding of the topic

std :: move()

Whenever you want to convert the lvalue to an rvalue you can use a function in the standard library 'std::move(object)'

You can learn from cpp reference or from here and here

The Move constructor

Since now we know about move semantics and the need of it, we will make a move constructor for the complex-numbers class.

```
// Move constructor
Complex(Complex &&obj)
{
    real = obj.real;
    imag = obj.imag;
    obj.real = nullptr;
    obj.imag = nullptr;
}
```

The Move Assignment

Similar to how we have a copy assignment we can have a move assignment too

```
Complex &operator=(Complex &&obj)
{
    real = obj.real;
    imag = obj.imag;
    obj.real = nullptr;
    obj.imag = nullptr;
    return *this;
}
```

The Rule of Big-5

Till now you know the rule of big-3 i.e. each class should have at-least these three functions:

- A default constructor
- A copy constructor
- A destructor

But now since we know about move semantics we would extend it and now say that in addition to these a class should have a move constructor and move assignment operator too

You can find the whole implemented Complex-numbers class here. As a programming exercise try to implement any class of your choice(like matrices etc) while following The Rule of Big-5 and overloading operators as necessary

Passing values to function by lvalue reference

Suppose you are calling a function anywhere in your program that requires a object to be passed and function may change the value of the object and you want the change to be reflected in the function where you called it.
There are two ways to solve this either use the object pointer or use lvalue reference as shown below :

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

void func(vector<int> *v){
    (*v)[0] = 10;
    (*v)[1] = 20;
    (*v)[2] = 30;
}
int main(){
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    func(&v);
    for(auto i : v)
        cout << i << " ";
}
```

Using pointer to object

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

void func(vector<int> &v){
    v[0] = 10;
    v[1] = 20;
    v[2] = 30;
}
int main(){
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    func(v);
    for(auto i : v)
        cout << i << " ";
}
```

Using lvalue reference

Iterating through a container using lvalue reference

We have already seen how easy it is to iterate through the elements of the iterator using for(auto i : v) But there is a problem in this technique whenever you iterate over the elements of v like this, for each iteration you are creating a copy of i, that is for each iteration i is just a copy of the element present at that position. Now this would be very costly if v is suppose a vector of vectors, then for each iteration we are creating a new copy of a vector. So again an efficient way to do is using lvalue referencing as follows :-

```cpp
vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    for(const auto &i : v)
        cout << i << " ";
```

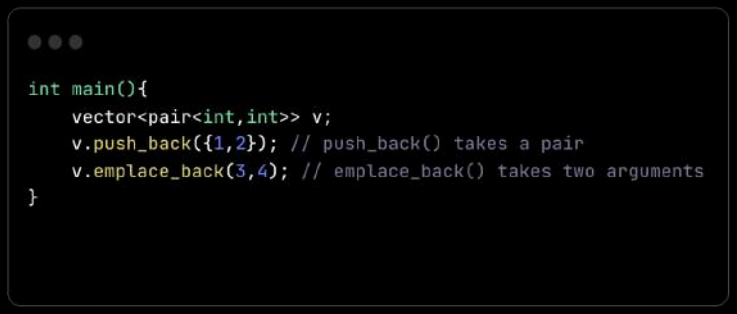
Here const is used just in case if we change the value of i our program warns us

push_back() vs emplace_back()

push_back() and emplace_back does the same thing of pushing an object at the end of a vector but there is a minor difference (which can have impact on the performance of your program). Suppose we have a vector of objects and we want to push an object(say obj1) in it. So when you call the push_back function, the program calls the copy constructor of the class to which the obj1 belongs thus creating a temporary object. This object is now copied to the end of the vector and then destroyed using a destructor. While on the other_hand emplace_back just take a list of arguments of the object creating an object at the end of the vector using the default constructor of the class thus saving time and space

Learn more about it here

Also see the following example for better understanding



```
int main(){
    vector<pair<int,int>> v;
    v.push_back({1,2}); // push_back() takes a pair
    v.emplace_back(3,4); // emplace_back() takes two arguments
}
```

v = { {1,2}, {3,4}}

Perfect Forwarding

Suppose you have a function whose input types(whether they are lvalue of rvalue references) are not known yet(so you use template while defining the function body) and inside the function body you call another function whose inputs are same as the given inputs but since we don't know the type of the inputs this may cause us to overload the first function for all possible inputs which may grow exponentially and is not a good way for programming

To prevent this, the standard template library provide us a function known as forward which forwards the argument to another function with the value category it had when passed to the calling function.

Look at this video or this article to understand the concept better

Day 2

Algorithms Library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as [first, last) where last refers to the element past the last element to inspect or modify.
This library can be included using #include<algorithm>
Some important functions in the library are:

- all_of, any_of, none_of: Find whether a predicate is true for all/any/none of the entries in a range. More: all_of , none_of and any_of
- for_each: Applies a function to all the entries in the range.More
- find: Finds the iterator to the first occurrence of an element in the range, if none is found, the function returns last. More
- find_if, find_if_not: finds the iterator to the first element which returns true/ false on the predicate respectively, if there is no such element, then the function returns last. More
- find_end: Finds the iterator to the last occurrence of an element/sequence in the range, if none is found, the function returns last.More
- count: Returns the number of elements in a range whose values match a specified value. More
- count_if: Returns the number of elements in a range whose values match a specified condition.More

- search: Searches for the first occurrence of a particular sequence in the range, and returns the iterator to the first element in the range, and last if no such sequence exists. More

- copy: Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction. More

- copy_backward: Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.More

- copy_if: Copy all elements in a given range that test true for a specified condition. More

- copy_n: Copies a specified number of elements.More

- fill: Assigns the same new value to every element in a specified range.More

- fill_n: Assigns a new value to a specified number of elements in a range starting with a particular element.More

- lexicographical_compare: Compares element by element between two sequences to determine which is lesser of the two.More

- lower_bound: Finds the position of the first element in an ordered range that has a value greater than or equivalent to a specified value, where the ordering criterion may be specified by a binary predicate.More

- max: Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate, similarly, the function min is defined.More, More

- max_element: Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate, similarly, the function min_element is defined. Note: the function minmax_element performs the work of both max_element and min_element in one call. More , More

- merge: Combines all the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.More

- minmax: Compares two input parameters and returns them as a pair, in order of least to greatest. More

- next_permutation: Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate, similarly, the function prev_permutation is defined.More, More

- partition: Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.More

- partial_sort: Arranges a specified number of the smaller elements in a range into a non-descending order or according to an ordering criterion specified by a binary predicate.More

- random _shuffle: Rearranges a sequence of N elements in a range into one of N! possible arrangements selected at random.More

- remove: Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.More

- remove_if: Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.More

- replace: Examines each element in a range and replaces it if it matches a specified value.More

- replace_if: Examines each element in a range and replaces it if it satisfies a specified predicate.More

- reverse:Examines each element in a range and replaces it if it satisfies a specified predicate.More
- sort:Arranges the elements in a specified range into a non-descending order or according to an ordering criterion specified by a binary predicate. More
- search:Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.More
- search_n:Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.More
- shuffle:Shuffles (rearranges) elements for a given range using a random number generator.More
- swap:Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.More
- unique:Removes duplicate elements that are adjacent to each other in a specified range.More
- upper_bound:Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.More

If you want to learn more about this amazing library, then check these articles:

- Article-1
- Article-2
- Article-3

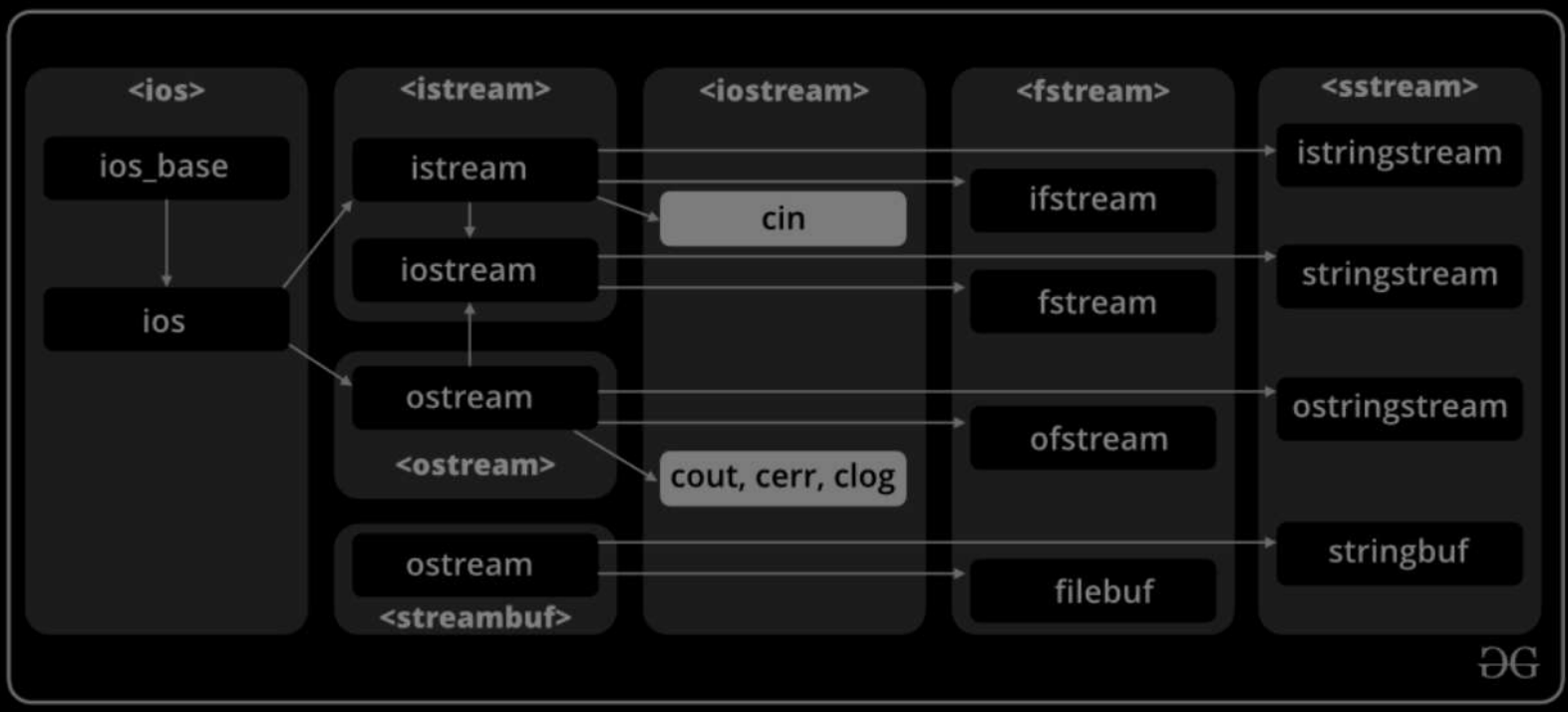
Day 3

Files and Streams

The I/O system of C++ contains a set of classes which define the file handling methods.
Just like how we use the iostream library which provides cin and cout methods for reading from standard input and writing to standard output respectively, there exists another standard C++ library called fstream (which has memeber classes ifstream, ofstream and fstream) , using which we can read and write from a file.

The classes are :-

1. ios - It stands for input output stream and is the base class for all other classes.
2. istream - It is derived from the ios class and stands for input stream. The extraction operator (>>) is overloaded in this class to handle input streams from files to the program execution.It also declares input functions such as get(), getline() and read().
3. ostream - It is also derived from the ios class and stands for output stream. The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution. It declares output functions such as put() and write().
4. fstreambase - This class provides operations common to the file streams and serves as a base for fstream, ifstream and ofstream class.
5. ifstream - This class represents the input file stream and is used to read information from files.
6. ofstream - This class represents the output file stream and is used to create files and to write information to files.
7. fstream - This class represents the file stream, and can be used to both read information from files and write information to files.
8. streambuf - This class contains a pointer which points to the buffer which is used to manage the input and output streams.
9. filebuf - Its purpose is to set the file buffers to read and write and it is also used to determine the length of the file.



<iostream> and <iomanip> header files

Like the cstdio header inherited from C's stdio.h, iostream provides basic input and output services for C++ programs. iostream uses the objects cin, cout, cerr, and clog for sending data to and from the standard streams input, output, error (unbuffered), and log (buffered) respectively. As part of the C++ standard library, these objects are a part of the std namespace.

iomanip on the other hand stands for input output manipulation and is used to set the properties of an iostream object. It includes important functions such as setprecision() , setbase() and put_time()

The Standard Input Stream (Cin)

The cin object in C++ is an object of class iostream. It is used to accept the input from the standard input device i.e. keyboard. It is associated with the standard C input stream stdin.

The extraction operator(>>) is used along with the object cin for reading inputs. The extraction operator extracts the data from the object cin which is entered using the keyboard.

Since it is an object it also contains many member function like cin.getline() and cin.read() Learn more about cin from this article

The Standard Output Stream(Cout)

The cout object in C++ is an object of class iostream. It is defined in iostream header file. It is used to display the output to the standard output device i.e. monitor. It is associated with the standard C output stream stdout.

The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).

Similar to cin it is also an object and contains member functions such as cout.put() and cout.write() etc. You can learn more about them here

The Standard Error Stream(Cerr) and Log Stream(Clog)

The cerr object in C++ is used to print error messages. It is defined in the iostream header file.

The clog object in C++ is an object of class iostream. It is associated with the standard C error output stream stderr.

clog and cerr, both are associated with stderr, but it differs from cerr in the sense that the streams in clog are buffered and not automatically tied with cout. We will talk about buffers and flushing in next section

Learn more about them here

We will talk more about them later

Stream Buffers and Flushing Streams

Stream buffers represent input or output devices and provide a low level interface for unformatted I/O to that device. Streams, on the other hand, provide a higher level wrapper around the buffer by way of basic unformatted I/O functions and especially via formatted I/O functions (i.e., operator<< and operator>> overloads). Stream objects may also manage a stream buffer's lifetime.

streambuf object helps to access these buffers.You can learn more about them in this article or this video

The buffer flush is used to transfer of computer data from one temporary storage area to computers permanent memory. If we change anything in some file, the changes we see on the screen are stored temporarily in a buffer.

Learn more here

Fast Input Output

Generally competitive programmers include these two lines at the starting of their main function to increase the speed of input and output. Here we explain their significance

- ios_base::sync_with_stdio(false);
  It toggles on or off the synchronisation of all the C++ standard streams with their corresponding standard C streams if it is called before the program performs its first input or output operation. Adding it changes the property sync_with_stdio to false (which is true by default). It is a static member of the function of std::ios_base.
- cin.tie(NULL);
  tie() is a method that simply guarantees the flushing of std::cout before std::cin accepts an input. This is useful for interactive console programs which require the console to be updated constantly but also slows down the program for large I/O. The NULL part just returns a NULL pointer.

endl vs '\n'

Endl and '\n' appears to do the same thing of adding a new line at the end of the output. The only major difference between them is that endl also flushes the output stream everytime it adds a '\n' which may make the program slow because flushing takes time

File Streams

In C++ we have a set of file handling methods. These include ifstream, ofstream, and fstream. These classes are derived from fstrembase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

```cpp
#include <fstream>
using namespace std;
int main()
{
        ifstream fin; // declares an object of type ifstream ofstream fout;
        ofstream fout; // declares an object of type ofstream
        fin.open( "infile.txt" );// for any stream object s, s.open() opens
                                 //filename and connects the stream s to it
                                 //so that chars can flow between them.
        fout.open( "outfile.txt" );
        int num1, num2, num3;
        fin >> num1 >> num2 >> num3;
        fout << "The sum is " << num1+num2+num3 << endl;
        fin.close();
        fout.close();
}
```

Whenever we open a file, it's good practice to check if opening the file was successful.

```cpp
in_stream.open("mydata");
if ( in_stream.fail() )
{
       cerr << "Could not open mydata.\n";
       exit(1);
       // 1 indicates an error occurred
}
```

Note - The fail() function works on any stream.
To use the exit() call, we need to include <cstdlib>

To append to a file, we can simply open the file in append mode in the open() member function

fout.open("outfile", ios::app);

To read more about the implementation of ifstream, ofstream, and fstream classes in File Handling,
Refer to this article

Stream Iterators

Stream iterators provide an iterator interface to the formatted extraction/ insertion operations of iostreams

Advancing the iterator performs one exraction, akin to std::cin >> n; if the extraction fails, the iterator assumes the singular state, which is also the state of the default-constructed iterator.

Output stream iterators can be used similarly to turn a stream into a "container", useful for algorithms that work with iterators

You can learn more about stream iterators from this article

For output iterators refer to this article

To learn more about istream_iterator refer here

Day 4

Memory Leak

Memory leakage occurs in C++ when programmers allocates memory by using new keyword and forgets to deallocate the memory by using delete() function or delete[] operator. If a program has memory leaks, then its memory usage increases very fast. Since all systems have limited amount of memory and memory is costly, it will create problems like reduced performance, program running out of memory and also some security vulnerabilities. However, you can prevent the issue of memory leak if you read this article. One of the way to avoid memory leak is to use smart pointers which we will cover below.

Smart Pointers

Smart Pointers are a wrapper to the normal pointers, designed to automatically managing them by performing memory allocation and deallocation as needed. It prevents memory leak, and makes it easier to control dynamically allocated objects in C++ programs. Smart pointers in C++ allows the user to have more control over the memory management compared to the Garbage collection mechanism in Java, making it easy to predict the performance of the program. You can learn more about smart pointers here.
The types of smart pointers are:

- unique_ptr - unique_ptr is a unique ownership smart pointer, meaning that it cannot be copied but can be moved. This ensures that there is only one owner of the object at any given time, avoiding any potential issues with multiple ownership and freeing of the same object.
- shared_ptr - shared_ptr supports copy semantics, so when a shared_ptr is copied, the reference count of the managed object is incremented, allowing multiple shared_ptrs to share ownership of the same object.
- weak_ptr - weak_ptr also points to an object managed by a shared_ptr, but it does not increment the reference count of the managed object.

The implementations of these smart pointers are covered in the previous video, however, you can also read this article for more information on them.
You can also read this article or this article which summarizes Smart Pointers.

Threading and Multithreading

Thread - Threads are lightweight units of execution that run within a process. They share the same memory space as the parent process, but have their own program counter and stack. We can implement threads using the following methods:

- Function Pointer
- Function Objects
- Lambda functions

This article shows the implementation of threads in C++ using the above methods or you can watch this video. You can also watch this video for more information.

Multi-threading - Multithreading refers to the parallel execution of multiple threads within a process. This allows the system to manage multiple tasks at the same time. This also allows for better performance on multi-core processors as threads could be distributed between cores, allowing better overall performance. You can read more about threads here.

join and detach

std::thread::join blocks the current thread until the specified thread finishes its execution whereas std::thread::detach separates the thread of execution from the current thread, allowing execution to continue independently. Any allocated resources will be freed once the thread exits. We could also use std::thread::joinable to check whether a thread could be joined or detached without any errors. To know more about the implementation watch this video or read this article.

Thread Synchronization

Thread synchronization in C++ is a critical aspect of multithreaded programming. It coordinates the execution of multiple threads to ensure that access to shared resources is properly managed and protected from race conditions. This is achieved through the use of synchronization mechanisms such as locks, semaphores, and monitors. Learn more about thread synchronization here.

Mutex

std::mutex is a synchronization primitive used to enforce mutual exclusion, allowing only one thread to access a shared resource at a time. It can be used to prevent race conditions, where multiple threads access and modify a shared resource simultaneously, leading to unexpected results. It provides two operation lock and unlock. The lock method in a std::mutex is used to acquire ownership of the mutex, allowing a thread to access a shared resource protected by the mutex whereas unlock allows access to other thread. You can read more about mutex here or watch this video.

Semaphore

A semaphore is a synchronization primitive used to control access to a shared resource in a multi-threaded environment. In C++, semaphores are typically implemented using the std::semaphore class, which is not a part of the standard library. Semaphore maintains a count of available resources, and allows or denies access to a shared resource based on the current count. When a thread requests access to the shared resource, it waits until the count is positive, and then decrements the count. When the thread is finished with the shared resource, it increments the count. The count acts as a limit on the number of threads that can access the shared resource simultaneously. There are two types of semaphore:

1. Binary Semaphore: It is mutex lock. It has two values 0 and 1.
2. Counting Semaphore: It is used to control access to a resource that has multiple instances. It has unrestricted domain.

You can learn more about semaphore by reading this article.

RAII

RAII stands for Resource Acquisition Is Initialisation. It is a technique used in C++ programming to ensure that resources, such as memory or file handles, are properly released when they are no longer needed. The basic idea is to tie the lifetime of a resource to the lifetime of an object, and to use the object's constructor and destructor to acquire and release the resource, respectively. This ensures that the resource will be properly released even in the event of an exception or a return from a function. You can learn more about RAII by watching this video or reading this article.

Day 5

Exception Handling in C++

Errors and Exceptions :

An extensive program or software rarely works the first time correctly. There may be errors in this.The two most common types of errors are:

1. Logical errors
2. Syntactic errors

These errors can be debugged by exhausting debugging and testing procedures. But programmers often encounter peculiar problems apart from logic or syntax errors. These types of errors are known as exceptions. Exceptions are runtime anomalies or unusual logical conditions that may come up while executing the C ++ program.

There are two types of Exceptions:

1. Synchronous exceptions: Errors such as out of range index and overflow.
2. Asynchronous exceptions: Exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.

What is Exception handling?

The main motive of the exception handling concept is to provide a means to detect and report an exception so that appropriate action can be taken. This mechanism needs a separate error handling code that performs the following tasks:

- Find and hit the problem (exception)
- Inform that the error has occurred (throw exception)
- Receive the error information (Catch the exception)
- Take corrective actions (handle exception)

Exception handling in C++ :

Exception handling in C++ consists of three keywords: try, throw and catch.

- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The throw keyword throws an exception when a problem is detected, which lets us create a custom error.
- The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

Note: The try and catch keywords always come in pairs.

Advantages of exception handling over traditional error handling in C++:

- An exception forces calling code to recognise an error condition and handle it. Unhandled exceptions stop program execution.
- An exception jumps to the point in the call stack that can handle the error. Intermediate functions can let the exception propagate. They don't have to coordinate with other layers.
- The exception stack-unwinding mechanism destroys all objects in scope after an exception is thrown, according to well-defined rules.
- An exception enables a clean separation between the code that detects the error and the code that handles the error.

A simple example to show exception handling in C++:

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

int main()
{
    int x = -1;
    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0){
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }
    cout << "After catch (Will be executed) \n";
    return 0;
}
```

Output :

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

Different Examples showing usage of try, catch and throw:

Exception Handling in C++ - GeeksforGeeks
C++ Exception Handling (tutorialspoint.com)
Exception Handling and Object Destruction in C++ - GeeksforGeeks

For deep dive in syntactical part of try:
try-block - cppreference.com

Runtime Errors in C++

This kind of errors are occurred, when the program is executing. As this is not compilation error, so the compilation will be successfully done. Since these errors are generated after your program is compiled successfully these are generally hard to find and to get the precise line where the error is occurring is found out using debuggers(For example GDB) which we will discuss later

Here are the examples of most commonly encountered runtime errors

Segmentation Fault(SIGSEGV)

Segmentation fault usually occurs in a C++ program when you try to access a memory which is does not belong to your process

It is one of the most commonly encountered error

You can see here the most common reasons that can lead to a segmentation fault error

Bus Error(SIGBUS)

On the other hand Bus error is caused when you are using a processor instruction with an address that does not satisfy its alignment requirements.

You can learn more about bus error in this article

Abort Signal (SIGABRT)

If an error itself is detected by the program then this signal is generated using call to abort() function. This signal is also used by standard library to report an internal error. assert() function in C++ also uses abort() to generate this signal.

You can learn about abort(), assert() and exit() in this article

You can learn more about SIGABRT from this discussion

Stack Overflow

All the functions are executed in a global stack in C++ so, naturally all the variables created inside these function also reside also lie in this stack so if we try to declare large variable inside a function that may lead to the overflow of stack

Another reason that may cause stack overflow is infinite recursion

Learn more about stack overflow here

Although you can set the stack size of your program using this preprocessor directive but this is not advisable as asking for memory more than that your RAM can give will lead to less space for operating system programs and can lead to unexpected shutdown of your system and repeatedly doing this may lead to some technical complications

```
#pragma comment(linker, "/STACK:2000000")
```

Heap Overflow

Although it is usually said that heap size is much more than that of stack size and we can allocate memory dynamically in heap freely, it is not true that there is nothing called heap overflow. If you keep allocating more and more memory in heap without freeing the memory which is of no use now it may lead to memory leak and would further lead to heap overflow. But this rarely happens because of efficient memory allocation in C++. Learn more from this answer

Cerr(Standard error stream)

This Stream is an unbuffered output stream so it is used when we need to display the error message immediately and does not store the error message to display later. Learn more about it in these articles Article 1 Article 2

You can also learn about cerr from cpp-reference

You may notice that by default the output of both cerr and cout goes to the terminal itself so why to use it in the first place instead of cout. The main reason is that it gives you more control over how you handle your error stream as might be the case that you are developing an application where you want to send all the error messages in a log file which can be then done using cerr

Debugging in C++

You can define your own debug method which you can use while writing your program in C++ as sometimes you want to see the values of the variables in the middle of the program

You can see one of the implementation in this codeforces blog

Debugging using GDB

GDB stands for GNU debugger tool and can be used for debugging C or C++ files. Read this article to learn basics of GDB

You can learn more about gdb from this article

VsCode also supports C++ debugging with a nice GUI interface which is efficient and easy to understand. Learn more about it here

Day 6

Strings in C++

String vs Character Array	
String	Char Array
A string is a **class that defines objects** that be represented as a stream of characters.	A character array is simply an **array of characters** that can be terminated by a null character.
In the case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated, **no memory is wasted**. **Strings are slower** when compared to implementation than character array.	The size of the character array has to be **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is also wasted** Implementation of **character array is faster** than std:: string.
String class defines **a number of functionalities** that allow manifold operations on strings.	Character arrays **do not offer** many **inbuilt functions** to manipulate strings.

Constructors:

- `string ()`
  - creates an empty string (`""`)
- `string ( other_string )`
  - creates a string identical to `other_string`
- `string ( other_string, position, count )`
  - creates a string that contains `count` characters from `other_string`, starting at `position`. If `count` is missing (only the first two arguments are given), all the characters from `other_string`, starting at `position` and going to the end of `other_string`, are included in the new string.
- `string ( count, character )`
  - create a string containing `character` repeated `count` times

The C++ strings library includes support for three types of strings:

- std::basic_string - a templated class designed to manipulate strings of any character type.
- std::basic_string_view (C++17) - a lightweight non-owning read-only view into a subsequence of a string.
- Null-terminated strings - arrays of characters terminated by a special null character.

For general purposes, we use std::string, which is a specialisation of std::basic_string.
The std:string class stores the characters as a sequence of bytes with the functionality of allowing access to the single-byte character.

Input operations on strings:
- getline() - Used to store a stream of characters entered by user in the object memory.
- push_back() - Used to input a character at the end of the string.
- pop_back() - Used to delete the last character from the string. (Introduced from C++11)

Capacity functions on strings:
- capacity() - Returns the capacity allocated to the string
- resize() - Changes the size of string (can increase or decrease)
- length() - Returns the length of string

- shrink_to_fit() - Decreases capacity of strings to its minimum capacity (useful to save memory if no further characters are added)

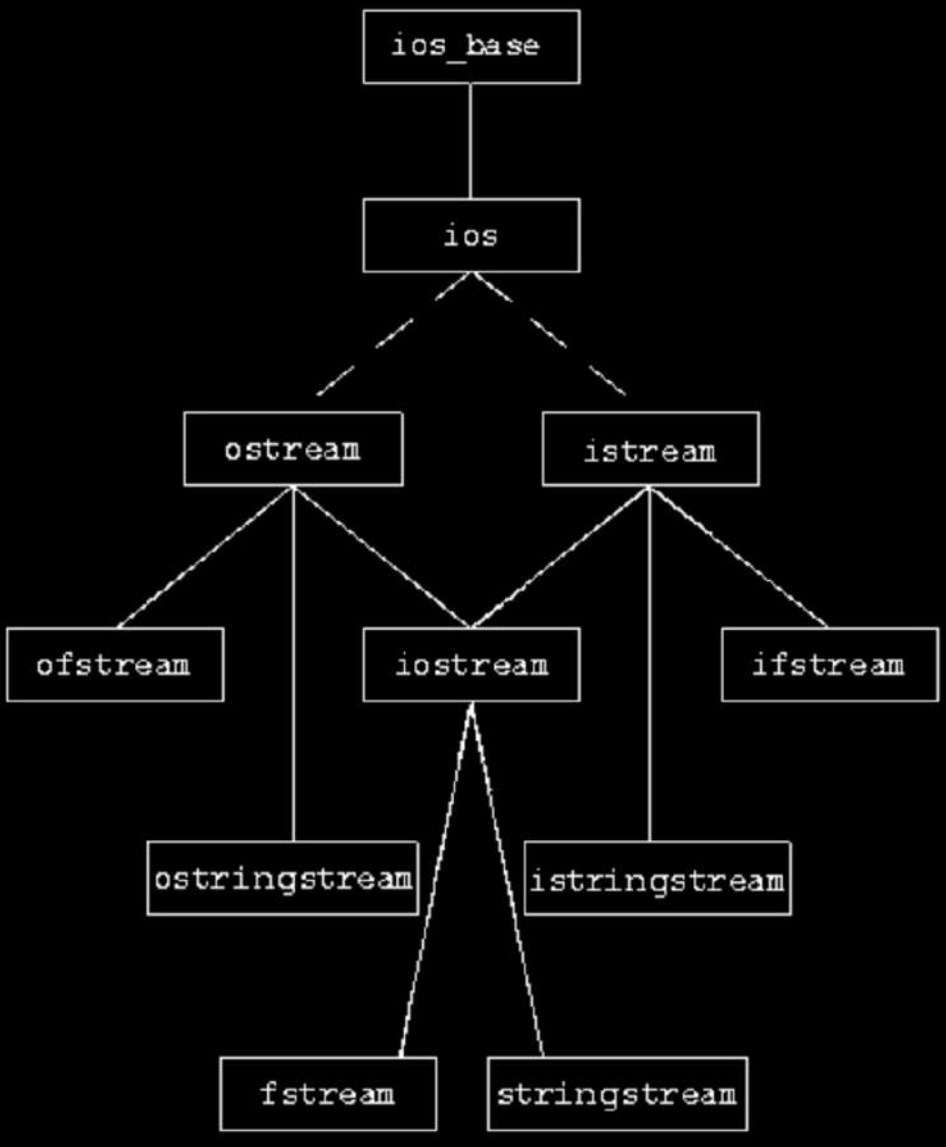
Iterator funcs on strings:
- begin() - Returns an iterator to the beginning of the string.
- end() - Returns an iterator to the next of the end of the string
- rbegin() - Returns a reverse iterator to the end of the string
- rend() - Returns an reverse iterator to the previous of the beginning of the string

Extra Material: Reading1, Reading2, Reading3, Video

Stringstream in CPP

A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin). To include this feature in our code, we need to include the sstream library.
In, sstream the standard library provides streams to and from a string:
• istringstreams for reading from a string
• ostringstreams for writing to a string
• stringstreams for reading from and writing to a string
It is very useful in parsing input. These operations are frequently used to convert textual data types to numerical data types and the other way around.
The istringstream type reads a string, ostringstream writes a string, and stringstream reads and writes the string.

Below is the inheritance diagram of the stringstream class:



To understand the difference between istringstream/ostringstream and string stream, read this discussion.

Performing the write operations on a stringstream

To perform the write operation in the stringstream object we have three methods present. Writing in stringstream means adding the data in the object of the stringstream class for which we can:

1. Using Constructor: While declaring the stringstream object by passing the string or any other data variable to the stringstream constructor.
2. Using insertion (<<) operator: We can use the insertion operation in the same way we use it with cout and can write data in stringstream object.
3. Using the str() function: As we have seen the str() function of the stringstream class, we can use it to insert any string value to the stringstream class object.

```cpp
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
    // performing write operation at the time of declaration

    string var = "Method1"; // creating a string first
    stringstream obj1(var); // declaring and initializing stringstream object
    cout<<obj1.str()<<endl;

    // Using the insertion operator

    stringstream obj2;
    obj2<<"Method2"; // adding value in obj2 using insertion operator
    cout<<obj2.str()<<endl;

    stringstream obj3;
    obj3.str("Method3"); // add value in obj3 using str() function
    cout<<obj3.str()<<endl;

    return 0;
}
```

Performing the read operations on a stringstream

To perform the read operation from the stringstream object we have two different methods are present. Reading from the stringstream means getting the data written in the object of the stringstream class for which we can:

1. Using extraction (>>) operator: We can use the extraction operation in the same way we use it with cin to get data from the input stream and can read data from the stringstream object.
2. Using the str() function: As we have seen the str() function of the stringstream class above in the article, we can use it to get data from the stringstream class object.

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
    // performing read operation using extraction operator
    stringstream obj1;

    // adding value in obj1 using insertion operator
    obj1<<"Method1 to read the data from stringstream object";
    string buf;
    while(obj1 >> buf)
    {
        cout<<buf<<" ";
    }
    cout<<endl;

    // Using str() method to get the data from the string
    stringstream obj2;

     // add value in obj2 using str() function
    obj2.str("Method2 to read the data from stringstream object");
    cout<<obj2.str()<<endl;

    return 0;
}
```

Day 7

Assembly Language

An assembly language is a type of low-level programming language that is intended to communicate directly with a computer's hardware. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans.

There is a general misconception among many people that there C/C++ code is first converted into an assembly code and then converted into machine level code but this is not true. Nowadays, compiler will just generate some assembly in memory but then convert it to machine code themselves and only write the machine code to file. Learn more

If you want to look at the assembly code for your C/C++ code run the following command in terminal

```
gcc -S filename.c
```

```
int main()
{
    int a = 2000, b =17;
}
```

```
.arch armv8-a
        .text
        .align  2
        .globl _main
_main:
LFB0:
        sub     sp, sp, #16
LCFI0:
        mov     w0, 2000
        str     w0, [sp, 12]
        mov     w0, 17
        str     w0, [sp, 8]
        mov     w0, 0
        add     sp, sp, 16
```

First few lines of this code's assembly

Writing assembly code directly in C++ code

You can directly write the assembly code in your C++ code using the asm declaration. You can learn more about it here

asm code for adding two numbers:

```
int add(int a, int b)
{
    int result;
    asm ( "addl %1, %2\n\t"
          "movl %2, %0"
          : "=r" (result)
          : "r" (a), "r" (b) );
    return result;
}
```

asm code for multiplying two numbers:

```
int multiply(int a, int b)
{
    int result;
    asm ( "movl %1, %%eax\n\t"
          "imull %2, %%eax\n\t"
          "movl %%eax, %0"
          : "=r" (result)
          : "r" (a), "r" (b) );
    return result;
}
```

Do try these on your system, try online compiler if not working. This may be because of use of different os. More examples here.