
C++:
LABYRINTH
OF
WONDERS
TANMAY
SHARMA

Acknowledgement & Copyright Letter

I would like to express my sincere gratitude to the following resources that have been invaluable in my learning journey of C++:

- GeeksforGeeks
- javatpoint
- cplusplus.com
- W3Schools

These websites have provided comprehensive tutorials, examples, and explanations, covering a wide range of C++ concepts and topics. Their clear and concise presentations have significantly enhanced my understanding of the language.

Email: velocitytanmay@gmail.com

Index

C++: Data Types

Data Type	Size	Description
`boolean`	1 byte	Stores true or false values
`char`	1 byte	Stores a single character/letter/number, or ASCII values
`int`	2 or 4 bytes	Stores whole numbers, without decimals
`float`	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
`double`	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

Modifiers

Struct

- Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.
- Unlike an array, a structure can contain many different data types (int, string, bool, etc.).
- Code

```
struct myDataType { // This structure is named "myDataType"
  int myNum;
  string myString;
};
```

- **Usecases:**
    - Structs can be used as user-defined datatypes
    - We can create variables of type struct
    - Encapsulation of data
    - Structs can behave like classes in C++, they can have their own constructors and destructors
    - We can define methods in structs in C++, NOT in C language

Enum

- An enum is a special type that represents a group of constants (unchangeable values).
- Default start from 0, but you can assign start value to starting constant.
- Code

```cpp
#include <bits/stdc++.h>
using namespace std;

enum week {
    monday = 1,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday,
    sunday
};

int main() {

    week today = wednesday;
    cout << today << endl;

    return 0;
}
```

- **Usecases:**
  - Heard of READ, WRITE, EXECUTE? That's where these are used
  - Code

```cpp
#include <bits/stdc++.h>
using namespace std;

enum permission {
    EXECUTE = 1,
    WRITE = 2,
    READ = 4
};

int main() {

    int permission = EXECUTE | READ | WRITE;
    cout << permission << endl;

    // chmod u=rwx,g=rx,o=r myfile
    // example "sudo chmod +777 etc/file.network"

    return 0;
}

// output
// 7
```

Union

- union is a user-defined datatype in which we can define members of different types of data types just like structures.
- But one thing that makes it different from structures is that the member variables in a union share the same memory location, unlike a structure that allocates memory separately for each member variable.
- The size of the union is equal to the size of the largest data type.
- Like structs unions can have members, constructors and destructors in C++, NOT in C.
- **Usecases:**
    - When the available memory is limited, it can be used to achieve memory efficiency.
    - It is used to encapsulate different types of data members.
    - It helps in optimizing the performance of applications.
- If you allocate values to all variables in a union, only the last one retains and others give garbage values
- Code

```cpp
#include<bits/stdc++.h>
using namespace std;

union myunion {
public:
    int data;
    int age;
    char grade;


    myunion(int d, int a, char g) {
        this->age = a;
        this->data = d;
        this->grade = g;
    }
};


int main(){

    return 0;
}
```

We can define anonymous struct, enum and union in a class,

```
#include <bits/stdc++.h>
using namespace std;

class student {
public:
    string name;

    struct {
        int standard;
        char grade;
    };

    enum {
        EXEC = 1,
        READ = 2,
        WRITE = 4
    };

    union {
        int enroll;
        int rank;
    };
};

int main() {

    return 0;
}
```

C++ Pointers

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

Definition of Pointers in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {

    int a = 12;
    int *addr = &a;

    cout << addr << endl;
    cout << *addr << endl;

    return 0;
}

// output
// 0x61ff08
// 12
```

*Why is there a need to specify data type for pointers as well? Since, it just stores an address then why specify data type? Why can't use int * instead of double * for a double data type?*

The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.

Pointer to a Pointer

```
#include <bits/stdc++.h>
using namespace std;

int main() {

    int a = 12;
    int *addr = &a;
    int **b = &addr;

    cout << b << endl;
    cout << *b << endl;
    cout << **b << endl;

    cout << endl;
    cout << addr << endl;
    cout << *addr << endl;

    return 0;
}

// Output
// 0x61ff04     // address of b
// 0x61ff08     // address of addr, through b
// 12           // value of a, through b

// 0x61ff08     // address of addr
// 12           // value of a, through addr
```

References and Pointers

There are 3 ways to pass C++ arguments to a function:

1. Call-By-Value
2. Call-By-Reference with a Pointer Argument
3. Call-By-Reference with a Reference Argument

```
#include <bits/stdc++.h>
using namespace std;

int find_val1(int n) {
    cout << "pass by value: " << n << endl;

    return 0;
}

int find_val2(int *n) {
    cout << "pass by reference, pointer: " << "address of n: " << n << " value
of dereferenced n: " << *n << endl;

    return 0;
}

int find_val3(int &n) {
    cout << "pass by reference, reference argument: " << "address of n: " << &n
<< " value of n: " << n << endl;

    return 0;
}

int main() {

    int n = 12;

    find_val1(n);       // pass by value
    find_val2(&n);      // pass by pointer
    find_val3(n);       // pass by reference

    return 0;
}
```

Call-By-Value

- **Mechanism**: A copy of the value of the argument is passed to the function.
- **Behavior**: Any modifications made to the argument within the function do not affect the original value outside the function.
- **Use Cases**: When you want to pass a value to a function without modifying the original variable.

Call-By-Reference with a Pointer Argument

- **Mechanism**: A pointer to the memory location of the argument is passed to the function.
- **Behavior**: Modifications made to the argument within the function directly affect the original value outside the function.
- **Use Cases**: When you want to modify the original value of an argument within a function.

Call-By-Reference with a Reference Argument

- **Mechanism**: A reference to the argument is passed to the function.
- **Behavior**: Modifications made to the argument within the function directly affect the original value outside the function.
- **Use Cases**: Similar to Call-By-Reference with a Pointer Argument, but often considered more convenient and safer.

What’s a reference?

Reference means alias. In simple words, we create another name for same variable. We can now perform operations on the same container having two or more names.

Array names are also pointers in C++

An array name can often be treated as a constant pointer to the first element of the array.

Are vectors too pointer names?
No, they aren't. C++ internally manages vectors by allocating memory dynamically. Vectors are dynamically resizable class, while arrays aren't class.

```
#include <bits/stdc++.h>
using namespace std;

void tanmay() {
    int arr[] = {1, 2, 3, 4, 5};
    cout << "accessing using normal method: " << endl;
    cout << arr[0] << " " << arr[1] << " " << arr[2] << endl
         << endl;

    // pointer to arr
    int *ptr = arr;

    cout << "accessing using pointer name dereferencing: " << endl;
    cout << *ptr << " " << *(ptr + 1) << " " << *(ptr + 2) << endl
         << endl;

    cout << "accessing using normal method for pointer name: " << endl;
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2] << endl
         << endl;
    cout << "accessing using array name dereferencing: " << endl;
    cout << *(arr) << " " << *(arr + 1) << " " << *(arr + 2) << endl;
}

int main() {

    tanmay();

    return 0;
}

// Output
// accessing using normal method:
// 1 2 3
```

```
// accessing using pointer name dereferencing:
// 1 2 3

// accessing using normal method for pointer name:
// 1 2 3

// accessing using array name dereferencing:
// 1 2 3
```

Feature	Array	Pointer
Declaration	data_type array_name[size];	data_type *pointer_name;
Memory Allocation	Static allocation (fixed size at compile time)	Dynamic allocation (size can be determined at runtime using new)
Access	array_name[index]	*(pointer_name + index)
Size	Fixed size at compile time	Can be changed at runtime
Arithmetic Operations	Pointer arithmetic can be applied to arrays	Pointer arithmetic can be applied directly
Initialization	Can be initialized directly with values	Must be initialized with an address
Passing to Functions	Passed by value (copy is made)	Passed by reference (original value is modified)
Common Use Cases	Storing collections of elements with known size	Dynamically allocating memory for unknown size collections, implementing linked lists, and other data structures

Array name are pointers to first element, but pointers and arrays are different

```cpp
#include <iostream>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Pointer to the first element of the array

    // Accessing elements:
    std::cout << arr[2] << std::endl;     // Array notation
    std::cout << *(ptr + 2) << std::endl; // Pointer arithmetic

    // Trying to modify the array name (invalid):
    // arr = new int[10]; // Error

    // Modifying the pointer:
    ptr = new int[10]; // Valid

    return 0;
}

// Output
// 3
// 3
```

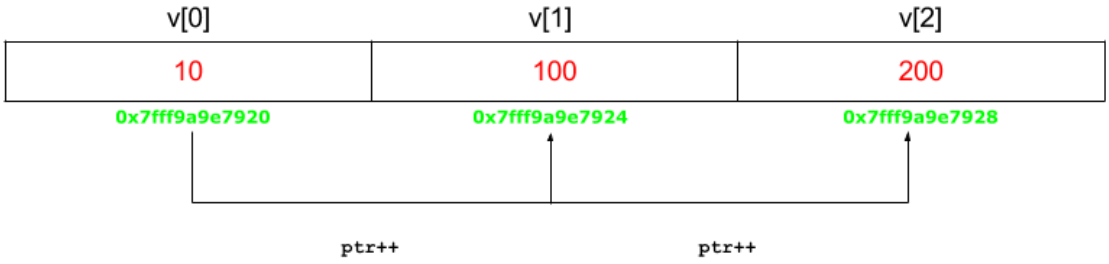
Size of pointer remains same for all data types (based on gcc compiler):
Even though ptr_arr points to arr, it's size is 4 bytes.

```
#include <bits/stdc++.h>
using namespace std;

int main() {

    int a = 0;
    char b = 'a';
    long c = 1l;
    double d = 0.0;
    long long e = 0L;

    int *ptra = &a;
    char *ptrb = &b;
    long *ptrc = &c;
    double *ptrd = &d;
    long long *ptre = &e;

    cout << sizeof(ptra) << endl
         << sizeof(ptrb) << endl
         << sizeof(ptrc) << endl
         << sizeof(ptrd) << endl
         << sizeof(ptre) << endl;

    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *ptr_arr = arr;

    cout << sizeof(arr) << endl
         << sizeof(ptr_arr) << endl;

    return 0;
}
```

Pointer Expressions and Pointer Arithmetic

- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)

Note: Pointer arithmetic is meaningless unless performed on an array.



Advanced Pointer Notation

```
int nums[2][3]  =  { { 16, 18, 20 }, { 25, 26, 27 } };
```

In general, nums[i][j] is equivalent to *(*(nums+i)+j)

Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Types of Pointers

String literals pointers

```
const char *str = "Hello World!";
cout << str << endl;
```

Void Pointers

- Void pointers have great flexibility as they can point to any data type. There is a payoff for this flexibility.
- These pointers cannot be directly dereferenced.
- They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

```
int a = 12;
void *ptr = &a;
int *next = (int *)ptr;

cout << ptr << " " << &ptr << " " << *next << endl;
```

Invalid pointers

- A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers.
- Uninitialized pointers are also invalid pointers.
- Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer. (Note: invalid pointers do not necessarily raise compile errors)

```
int *ptr1;
int arr[10];
int *ptr2 = arr + 20;
```

NULL Pointers

A null pointer is a pointer that point nowhere and not just an invalid address. Following are 2 methods to assign a pointer as NULL.

```
int *ptr1 = 0;
int *ptr2 = NULL;
```

Dangling Pointer

- A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
- Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

```
int *ptr = (int *)malloc(sizeof(int));

// After below free call, ptr becomes a dangling pointer
free(ptr);
printf("Memory freed\n");

// removing Dangling Pointer
ptr = NULL;
```

Reasons for dangling pointer

- Memory deleted
- Function returning pointer of local variable
- When variable goes out of scope, eg, defining pointer within small scope

Wild Pointer

- A pointer that has not been initialized to anything (not even NULL) is known as a wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

  ```
  dataType *pointerName;
  ```

```
Segmentation Fault (SIGSEGV)
timeout: the monitored command dum
/bin/bash: line 1:    32 Segmentat
```

Advantages of Pointers

- Pointers reduce the code and improve performance. They are used to retrieve strings, trees, arrays, structures, and functions.
- Pointers allow us to return multiple values from functions.
- In addition to this, pointers allow us to access a memory location in the computer's memory.
- Functions also have addresses in memory
- Function names too acts as pointers to their addresses in memory

```cpp
#include <bits/stdc++.h>
using namespace std;

void tanmay() {
    cout << "Tanmay S." << endl;
}

int main() {

    printf("%p\n", main);
    printf("%p\n", tanmay);

    return 0;
}

// Output
// 0040148E
// 00401460
```

Function Pointers

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```
#include <bits/stdc++.h>
using namespace std;

void computer(int n) {
    cout << "value of n: " << n << endl;
}

int main() {

    void (*ptr)(int) = &computer;

    ptr(10);    // either this way
    (*ptr)(11); // or this way both ways it works

    return 0;
}
```

More Knowledge about Pointers

Array of Function Pointers

- Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.
- These can be used in place of switch case statements.
- You can pass pointers to a function as well, and pointer functions in another function as well

```cpp
#include <bits/stdc++.h>
using namespace std;

void add(int a, int b) {
    cout << "addition: " << (a + b) << endl;
}

void subtract(int a, int b) {
    cout << "subtraction: " << (a - b) << endl;
}

void multiply(int a, int b) {
    cout << "multiplication: " << (a * b) << endl;
}

int main() {

    void (*fun_ptr[])(int, int) = {&add, &subtract, &multiply};
    int a = 12, b = 5;
    (*fun_ptr[0])(a, b);
    (*fun_ptr[1])(a, b);
    (*fun_ptr[2])(a, b);

    return 0;
}

// Output
// addition: 17
// subtraction: 7
// multiplication: 60
```

Functions returning pointers

```
#include <bits/stdc++.h>
using namespace std;

// instead make x static or global
int *func() {
    int x = 100;
    int *ptr = &x;

    return ptr;
}
int main() {
    int *addr = func();
    fflush(stdin);

    printf("%p\n", addr);

    return 0;
}

// Output
// 0061FEF0
```

Difference between pointer to constant, constant pointer and constant pointer to constants

Pointer to Constant

Here, the pointed variable is constant, it can't change, but the pointer can change and point to another const variable

```
const int high = 100;
const int *ptr = &high;
cout << high << " " << *ptr << endl;

const int low = 0;
ptr = &low;
cout << low << " " << *ptr << endl;
```

Constant Pointer

This pointer can only hold single address, when tried to change to another memory, it gives read-only variable error

```
int high = 100;
int *const ptr = &high;

cout << high << " " << *ptr << endl;

int low = 0;
ptr = &low; // gives error, read-only variable 'ptr'
cout << low << " " << *ptr << endl;
```

The value residing in *ptr can be changed, but not memory

```
*ptr = 1000;
cout << *ptr << endl
     << ptr << endl;
```

Constant pointer to constant

Constant value and constant memory in pointer

```
const int high = 100;
const int low = 0;

const int *const ptr = &high;
cout << high << " " << *ptr << endl;

ptr = &low;  // gives error, read-only variable 'ptr'
*ptr = 1000; // assignment of read-only location error
```

Malloc, Calloc, Realloc and Free

Header file required: `<stdlib.h>`

1. Malloc (Memory Allocation)
2. Calloc (Contiguous Memory Allocation)
3. Realloc (Reallocation)
4. Free (Memory Deallocation)

Malloc

- Memory allocation is used to assign a large chunk of memory and returns a void pointer to it's base address. It just gives us a large chunk of memory, do whatever you want with it.
- It takes only size of memory as argument

```
// gives me a memory block of 4 * 4 = 16 size (as per gcc)
int *ptr_malloc = (int *)malloc(4 * sizeof(int));

for (int i = 0; i < 4; i++) {
    ptr_malloc[i] = i + 1;
}

for (int i = 0; i < 4; i++) {
    cout << ptr_malloc[i] << " ";
}
cout << endl;
```

Calloc

- Does the same but in a contiguous manner and asks for number of memory blocks with size of each block.
- It takes number of blocks and size of each blocks as arguments
- In following case, I've used memory using int but assigned to a char pointer

```
// gives me 5 block of size 4 each = 20 bytes (as per gcc)
// since I'm storing character here, of 1 byte each
// it only uses 1 byte
char *ptr_calloc = (char *)calloc(5, sizeof(int));

for (int i = 0; i < 5; i++) {
    ptr_calloc[i] = 'a' + i;
}

for (int i = 0; i < 5; i++) {
    cout << ptr_calloc[i] << " ";
}
cout << endl;

for (int i = 0; i < 5; i++) {
    printf("%p %d\n", (ptr_calloc + i), sizeof(ptr_calloc[i]));
}
cout << endl;
```

Free

Frees up the space or deallocates the space given using malloc or calloc.

```
free(ptr_calloc);
```

Realloc

This one is used to change the size of memory issued using malloc or calloc at runtime, it can increase the size or decrease the size.

```
ptr_malloc = (int *)realloc(ptr_malloc, 10 * sizeof(int));

for (int i = 4; i < 10; i++) {
    ptr_malloc[i] = i + 1;
}

for (int i = 0; i < 10; i++) {
    cout << ptr_malloc[i] << " ";
}
cout << endl;
```

Pointer Usecases

- Dynamic memory allocation
- Used in data structures like linked lists, trees and graphs where we don't know the final size of data
- Used in low level memory operations for writing device drivers
- Function pointers are used in interrupt handling and callback mechanisms (in real world you don't see all interrupts coming up at once, they're executed sequentially one after the other and OS doesn't create instances of them, it simply uses pointers and addresses to call those interrupt handlers by reference)
- Used in accessing array elements, array of any dimension

Are Pointers Integers?

No, they're not. Pointer is an address and a positive hexadecimal number.

Ways to initialize pointers?

Pointer variables are initialized by one of the following ways.

- Static Memory Allocation

```
int data = 10;
int *ptr = &data; // Pointer to a statically allocated variable
```

- Dynamic Memory Allocation

```
int *ptr = (int *)malloc(sizeof(int)); // Allocate memory for an integer
if (ptr != nullptr) {
    *ptr = 20; // Assign a value to the allocated memory
}
```

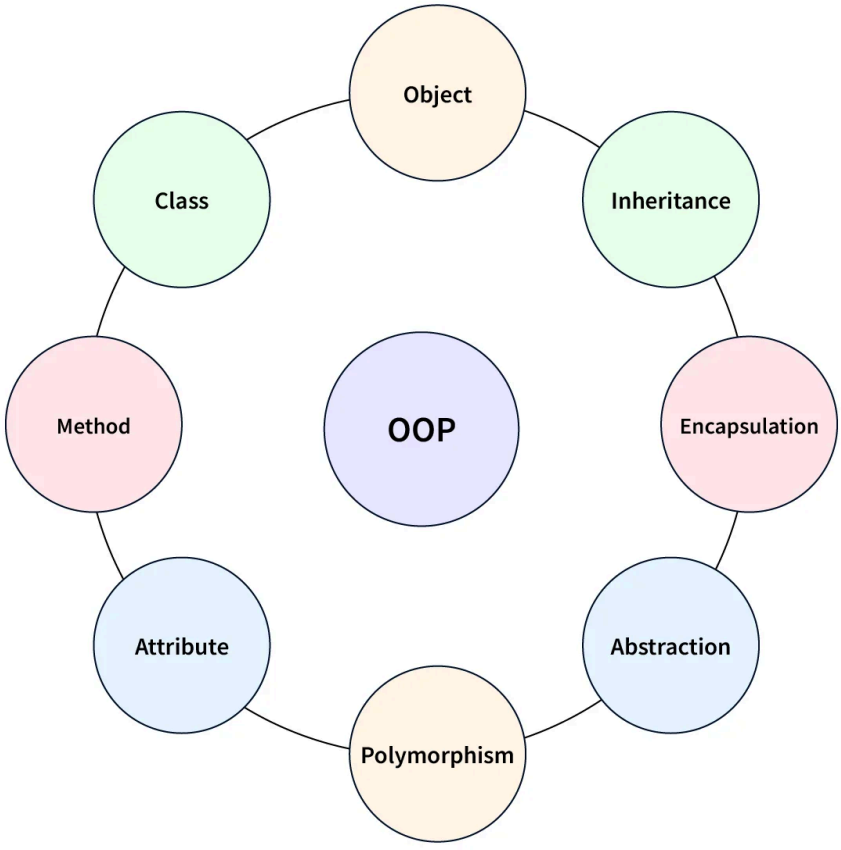
C++ OOP

OOP

Object Oriented Programming is a paradigm of computer programming that utilize real-world entities to make programs easy to understand to humans. It gives the code an intuitive reasoning to bee more realistic and be a part of natural thinking of human.

There are some basic concepts that act as the building blocks of OOPs i.e.

- Class
- Objects
- Encapsulation
- Abstraction
- Polymorphism
- Inheritance
- Dynamic Binding
- Message Passing



Class

- A class is the building block in whole of the OOPs.
- It is a user-defined data type and
- It's like a blueprint for an object that contains the features and behaviours.
- Here, features are the data members and behaviours are member functions of that class.
- Member functions are defined to manipulate these data members
- These members and functions can only be accessed by creating an instance of that class.
    - For eg, cars are available in every type everywhere hence we can consider a class of cars with common properties, where
        - Each car has 4 tyres, their average, max speed, shift mechanism, etc
        - And each car has functions like increase_speed, apply_break, etc

```
class Cars {
public:
    int wheels = 4;
    int doors;
    float speed;
    double mileage;
    double tank_size_in_ltr;

    Cars(int drs, double mlg, double tank_size) {
        this->doors = drs;
        this->mileage = mlg;
        this->tank_size_in_ltr = tank_size;
    }

    void increase_speed(float acceleration, float time) {
        this->speed += (acceleration * time);
    }
};
```

Object

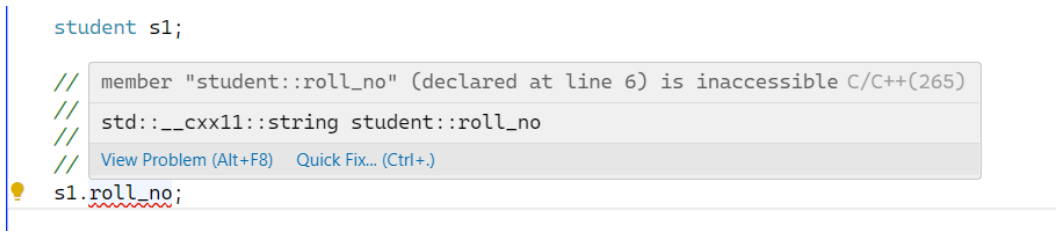
- An object is an instance of class, like an entiry having features and behaviours of that class
- Class itself doesn't occupy memory on it's own, but when it's instance is created (i.e., the object), the object occupies memory
- Objects are used to carry information and communicate within the program or functionality

```
Cars mustang(2, 16.5, 42.7);
mustang.speed = 12;
float time = 12.2;
float acceleration = 18.2;

mustang.increase_speed(acceleration, time);
cout << mustang.speed << endl;
```

Access Modifiers

- Access modifiers control the access of members and function of a class.
- They're keywords and members and functions under them have that level of access.
- They are of three types:
    - **Public**
        - Public members can be accessed outside the class
    - **Private**
        - Private members can be accessed only inside the class
        - Either by member functions of friend functions, not even by classes inside base class
    - **Protected**
        - Protected members can be accessed within the class and the derived class (even in multilevel inheritance).
        - Not even the inner class members can access protected members of base class.

```cpp
#include <bits/stdc++.h>
using namespace std;

class student {
private:
    string roll_no;

public:
    string name;
    int standard;
    char division;

    void set_roll(string roll) {
        this->roll_no = roll;
    }

    void get_roll() {
        cout << "Roll no. is: " << this->roll_no << endl;
    }
};

int main() {

    student s1;

    // cannot access as it's private and
    // can be accessed inside the class only
    // that's why set_roll() and get_roll() are
    // able to access and manipulate it
    s1.roll_no;

    return 0;
}
```



Friend Class

- Friend class helps you access the private and protected members & functions of other classes.
- But they have to have a declaration inside the base class.
- Placement of a friend class doesn't get affected even if it's defined under public, private or protected, when declared inside base class.

```
#include <bits/stdc++.h>
using namespace std;

class accounts {
private:
    int customer_id;
    int bank_balance = 0;
    int loan_amount = 0;

public:
    accounts(int id, int balance, int amount) {
        this->customer_id = id;
        this->bank_balance = balance;
        this->loan_amount = amount;
    }

    friend class loans;
};

class loans {
public:
    void display(accounts &customer) {
        cout << endl
             << endl;

        cout << "Customer ID: " << customer.customer_id << endl
             << "Customer Balance: " << customer.bank_balance << endl
             << "Customer loan amount: " << customer.loan_amount << endl;

        cout << endl
             << endl;
    }
};
```

```
int main() {

    accounts acc1(12345, 10000, 5000);
    loans loan_acc1;
    loan_acc1.display(acc1);

    return 0;
}

// Output
// Customer ID: 12345
// Customer Balance: 10000
// Customer loan amount: 5000
```

Functions & Types

Member Functions

- Member functions can be defined inside the class and outside too.
- Before defining them outside you've to declare them inside the class, otherwise this won't work.
- To define them outside, we'll use scope resolution operator considering class name.

```cpp
#include <bits/stdc++.h>
using namespace std;

class student {
private:
    string roll_no;

public:
    string name;
    int standard;
    char division;

    void set_roll(string); // declaring set_roll()
    void get_roll();       // declaring get_roll()
};

// defining set_roll() and get_roll()
// outside the scope of class student
void student::set_roll(string roll) {
    this->roll_no = roll;
}
void student::get_roll() {
    cout << "Roll no. is: " << this->roll_no << endl;
}

int main() {

    student s1;

    s1.name = "Tanmay";
    cout << s1.name << endl;

    s1.set_roll("ABCX1211");
    s1.get_roll();
```

```
    return 0;
}
```

Inline Functions

- Inline functions are the functions which the compiler doesn't calls, it simply replaces the function calls with actual code.
- More like whenever you'll define inline functions it'll not call the function **BUT** replace the code in execution with inline function everytime it calls it.
- Inline functions can be useful when you have better code than compiler for an operation, thus improving performance.
- These can be defined using an inline keyword before function return-type.

Friend Functions

- Just like friend classes, friend functions can help access and manipulate private and protected members of a class.
- A function can be friendly with multiple classes.
- There are two ways to define a friend function:
    - Global Friend function
        - Declared inside base class and defined outside base class

```
#include <bits/stdc++.h>
using namespace std;

class base {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    base(int prvt, int prtc) {
        this->private_variable = prvt;
        this->protected_variable = prtc;
    }

    friend void display(base &obj);
```

```
};

void display(base &obj) {
    cout << endl
         << endl;

    cout << "Private variable: " << obj.private_variable << endl
         << "Public variable: " << obj.protected_variable << endl;

    cout << endl
         << endl;
}

int main() {

    base ball(101, 111);
    display(ball);

    return 0;
}
```

- Friend member function of another class

```
#include <bits/stdc++.h>
using namespace std;

// declaration before to avoid error in another class
// as we're using it's object in display function
class base;

// another class definition
class another {
public:
    void display(base &obj);
};

// base class definition
class base {
private:
    int private_variable;

protected:
    int protected_variable;
```

```
public:
    base(int prvt, int prtc) {
        this->private_variable = prvt;
        this->protected_variable = prtc;
    }

    // declaration of display function
    friend void another::display(base &obj);
};

// definition of display function
// look we did defined it the way we define functions whose
// definition is outside
// class_name::function_name
void another::display(base &obj) {
    cout << endl
         << endl;

    cout << "Private variable: " << obj.private_variable << endl
         << "Public variable: " << obj.protected_variable << endl;

    cout << endl
         << endl;
}

int main() {

    base ball(101, 111);

    another game;
    game.display(ball);

    return 0;
}
```

Advantages of Friend Functions

- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions

- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- Friend functions cannot do any run-time polymorphism in their members.

Important Points About Friend Functions and Classes

- Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming.
- Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- Friendship is not inherited.
- The concept of friends is not in Java.

Why do we give semicolons at the end of class?

- The main reason why semi-colons are there at the end of the class is compiler checks if the user is trying to create an instance of the class at the end of it.
- Yes, just like structure and union, we can also create the instance of a class at the end just before the semicolon.
- As a result, once execution reaches at that line, it creates a class and allocates memory to your instance.

```
#include <bits/stdc++.h>
using namespace std;

class student {
private:
    string roll_no;

public:
    string name;
    int standard;
    char division;

    void set_roll(string);
    void get_roll();

    student() {
        cout << "Hey student entry created!" << endl;
    }

} student_one; // object instantiated

int main() {

    return 0;
}
```

Constructors

- Constructor in C++ is a special method that is invoked automatically at the time an object of a class is created.
- It is used to initialize the data members of new objects generally.
- The constructor in C++ has the same name as the class or structure.
- It constructs the values i.e. provides data for the object which is why it is known as a constructor.
- They are of 4 types:
    - **Default Constructor**: No parameters. They are used to create an object with default values.
    - **Parameterized Constructor**: Takes parameters. Used to create an object with specific initial values.
    - **Copy Constructor**: Takes a reference to another object of the same class. Used to create a copy of an object.
    - **Move Constructor**: Takes an rvalue reference to another object. Transfers resources from a temporary object.

Default Constructors

- A constructor without any arguments or with the default value for every argument is said to be the Default constructor.
- By default the compiler defines the default constructor itself.
- If you define an explicit constructor with parameters, then declare an empty constructor with no parameters (for objects with no parameters).

Parameterized Constructors

- These constructors have parameters in their definition.
- Used for overloading default constructors.
- Used for assigning values to members.

Copy Constructors

- These constructors take the object of their class as parameter and their purpose is to copy the contents of one object to another.
- By default like other constructors they too exist and created by compiler to copy one object to another.
- It makes shallow copy, where all the references are kept as it is.

- In case of pointers, if the first object has its pointers deleted then the second object's pointers will become dangling pointers as their reference has been deleted.
- To make deep copy, where even pointers's copy is created, we need copy constructors.

Move Constructors

- Creates a new object by transferring resources from an existing object.
- Steals resources from the original object, leaving it in a valid but unspecified state.
- Generally more efficient than the copy constructor as it avoids unnecessary data copying.
- It makes a deep copy from initial object.

Destructors

- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence, destructor cannot be overloaded.
- It cannot be declared static or const.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

When destructor is called?

- Destructor is called when the function ends.
- Destructor is called when the program ends.
- Destructor is called when a block containing local variables ends.
- Destructor is called when a delete operator is called.

How to call destructors explicitly?

- Destructor can also be called explicitly for an object.

```cpp
#include <bits/stdc++.h>
using namespace std;

class student {
private:
    string roll_no;

public:
    string name;
    int standard;
    char division;

    void set_roll(string);
    void get_roll();

    student() {
        cout << "Hey student entry created!" << endl;
    }

    // destructor
    ~student() {
        cout << "destructor called!" << endl;
    }

} student_one; // object instantiated

int main() {

    student s1;

    // explicit destructor calling
    s1.~student();

    return 0;
}
```

Frequently Asked Questions on C++ Constructors & Destructors

What Are the Functions That Are Generated by the Compiler by Default, If We Do Not Provide Them Explicitly?

The functions that are generated by the compiler by default if we do not provide them explicitly are:

- Default Constructor
- Copy Constructor
- Move Constructors
- Assignment Operator
- Destructor

Can We Make the Constructors Private?

Yes, in C++, constructors can be made private. This means that no external code can directly create an object of that class.

How Constructors Are Different from a Normal Member Function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Default Constructors don't have input argument however, Copy and Parameterized Constructors have input arguments
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

Can We Have More Than One Constructor in a Class?

Yes, we can have more than one constructor in a class. It is called Constructor Overloading.

Can destructor be private?

Yes, destructor can be defined as private when we want to control the deletion of the object manually.

Static Keyword

Static Members

Static data members are class members that are declared using static keywords. A static member has certain special characteristics which are as follows:

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is created, even before the main starts outside the class itself.
- It is visible can be controlled with the class access specifiers.
- Its lifetime is the entire program.

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

// creating a dummy class to define the static data member
// it will inform when its type of the object will be
// created
class stMember {
public:
    int val;
    // constructor to inform when the instance is created
    stMember(int v = 10) : val(v) {
        cout << "Static Object Created" << endl;
    }
};

// creating a demo class with static data member of type
// stMember
class A {
public:
    // static data member
    static stMember s;
    A() { cout << "A's Constructor Called " << endl; }
};

stMember A::s = stMember(11);

// Driver code
int main() {

    // Statement 1: accessing static member without creating
    // the object
```

```
    cout << "accessing static member without creating the "
            "object: ";
    // this verifies the independency of the static data
    // member from the instances
    cout << A::s.val << endl;
    cout << endl;

    // Statement 2: Creating a single object to verify if
    // the seperate instance will be created for each object
    cout << "Creating object now: ";
    A obj1;
    cout << endl;

    // Statement 3: Creating multiple objects to verify that
    // each object will refer the same static member
    cout << "Creating object now: ";
    A obj2;
    cout << "Printing values from each object and classname"
         << endl;

    cout << "obj1.s.val: " << obj1.s.val << endl;
    cout << "obj2.s.val: " << obj2.s.val << endl;
    cout << "A::s.val: " << A::s.val << endl;

    return 0;
}
```

FAQs on static members?

Can static data members be private?
Yes, static data members can be private. They follow the same access control rules as regular data members and can be accessed through public member functions of the class.

Can we initialize a static data member within the class definition?
No, static data members must be defined outside the class definition. However, they can be initialized inline if they are of integral or enumeration type (C++17 onwards).

Can we inherit static data members ?
Static data members are class-specific and cannot be inherited. But we can access the static data member in the derived class directly using variable name

Can we have static member functions in a class?
Yes, static member functions can be defined in a class. They can access static data members but cannot access non-static data members or this pointer.

Static Member Function

Static Member Function in a class is the function that is declared as static because of which function attains certain properties as defined below:

- A static member function is independent of any object of the class.
- A static member function can be called even if no objects of the class exist.
- A static member function can also be accessed using the class name through the scope resolution operator.
- A static member function can access static data members and static member functions inside or outside of the class.
- They cannot access non-static members.
- Static member functions have a scope inside the class and cannot access the current object pointer.
- You can also use a static member function to determine how many objects of the class have been created.

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

class Box {
private:
    static int length;
    static int breadth;
    static int height;
    int hello;

public:
    static void print() {
        cout << "The value of the length is: " << length << endl;
        cout << "The value of the breadth is: " << breadth << endl;
        cout << "The value of the height is: " << height << endl;
    }
};

// initialize the static data members
int Box ::length = 10;
int Box ::breadth = 20;
int Box ::height = 30;

int main() {

    Box b;
```

```
    cout << "Static member function is called through Object name: \n"
         << endl;
    b.print();

    cout << "\nStatic member function is called through Class name: \n"
         << endl;
    Box::print();

    return 0;
}
```

‘this’ Pointer

- The this pointer is a special pointer that implicitly exists within the scope of a non-static member function of a class. It points to the current object, i.e., the object whose member function is being called.

How it works

- When a member function is invoked on an object, the compiler passes the address of that object as a hidden argument to the function.
- This hidden argument is the this pointer.

When is it used?

- **Accessing current object's members**: It's used to differentiate between local variables and member variables with the same name.
- **Returning the current object**: It's used in method chaining or to return a reference to the current object for further manipulation.
- **Passing the current object as an argument**: You can pass the this pointer as an argument to other functions.
- **Operator overloading**: It's used in operator overloading to manipulate objects.

```cpp
#include <iostream>

class MyClass {
public:
    int x;

    MyClass(int val) : x(val) {}

    void display() {
        std::cout << "x: " << x << std::endl;
    }

    MyClass &add(int val) {
        x += val;
        return *this; // Returning a reference to the current object
    }
};

int main() {
    MyClass obj(10);
    obj.display(); // Output: x: 10
```

```
    // method chaining
    obj.add(5).add(2).display(); // Output: x: 17
    return 0;
}
```

Scope Resolution Operator

- Only used for resolution of static members.
- If you want to use class member, then use *'this'.*
- *Scope resolution* and *'this'* are used for differentiating between class members and explicit variables.

```
#include <bits/stdc++.h>
using namespace std;

class Test {
public:
    static int a;

    void func(int a) {
        // scope resolution for member 'a'
        cout << Test::a << endl;
    }
};

int Test::a = 10;

int main() {

    Test t1;
    t1.func(12);

    return 0;
}
```

Local Classes

- A class declared inside a function becomes local to that function and is called Local Class in C++.
- A local class name can only be used locally i.e., inside the function and not outside it.
- The methods of a local class must be defined inside it only.
- A local class can have static functions but, not static data members.
- Local classes can access global types, variables, and functions.

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

void fun() {
    class Test { // local to function
    public:
        // Fine as the method is defined
        // inside the local class
        void method() {
            cout << "Local Class method() called";
        }
    };

    Test t;
    t.method();
}

int main() {
    fun();
    return 0;
}
// Output
// Local Class method() called
```

Nested Classes

- Nested classes are the classes which are declared inside other class.
- It might be possible to declare them outside but definition should be present.

```
#include <bits/stdc++.h>
using namespace std;

class Enclosed {
public:
    int x = 10;
    static const int z = 22;

    class Nested1; // nested class declared here

    class Nested2 { // nested2 class defined here
    public:
        int xy = 111;
    };

    void enc_func(Nested2 *n) {
        cout << "Called from outside Nested2, inside Enclosed: " << n->xy <<
endl;
    }
};

// nested class defined
class Enclosed::Nested1 {
public:
    int y = 11;

    void func(Enclosed *e) {
        cout << "Called from inside Nested1: " << this->y << endl;
        cout << "Enclosed variable x: " << e->x << endl
             << endl;
    }
};

int main() {

    Enclosed e1;
    Enclosed::Nested1 n1;
    Enclosed::Nested2 n2;
```

```
    n1.func(&e1);
    e1.enc_func(&n2);

    return 0;
}

// Output
// Called from inside Nested1: 11
// Enclosed variable x: 10

// Called from outside Nested2, inside Enclosed: 111
```

Difference between Classes and Structures

Feature	Class	Structure
Default Access Specifier	Private	Public
Members	Can have both data members and member functions	Can have both data members and member functions
Inheritance	Supports multiple inheritance	Supports multiple inheritance
Constructors and Destructors	Can have constructors and destructors	Can have constructors and destructors
Polymorphism	Supports polymorphism (virtual functions)	Supports polymorphism (virtual functions)
Common Use Cases	Modeling real-world entities, encapsulating data and behavior	Grouping related data together for organization

Inheritance

Inheritance is the process in which one object can acquire all the properties of its parent object automatically. It is used so that you can reuse, modify and extend the attributes and behaviours which are defined in another class.
Here, the class which inherits the attributes and behaviours is called the derived class and the one which is being inherited is called as base class.

Why use inheritance?
Inheritance is useful for code reusability. Now you can access the members and functions of parent class without writing the same code again.
For eg, there is a parent class named Vehicle containing required attributes like doors, tires, colors. You can extend this class to any vehicle type, be it car, bicycle, motorcycle, truck, etc.

C++ supports **five types of inheritance:**

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

Syntax of derived class,

```
class derived_class_name : access_modifier base_class_name {
    // body of derived_class
}
```

Derived_class_name: it is the name of derived class.

Access_modifier: it is the visibility mode which specifies whether we want to inherit features in public manner of private.

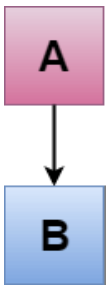
- When a class is inherited publicly,
    - Member Functions can access public & protected variables and functions inside the derived class.
    - Objects of derived class can access only the public variables and functions.
- When a class is inherited in protected mode,
    - Member functions can access only the public & protected variables and functions inside the derived class.
    - Objects of derived class cannot access any type of data.
- When a class is inherited in private mode,

- Member functions can access only the public & protected variables and functions inside the derived class.
- Objects of derived class cannot access anything.

Base_class_name: it is the name of the class which is being inherited.

By default, the access mode is private if you don't specify anything.

Single Level Inheritance



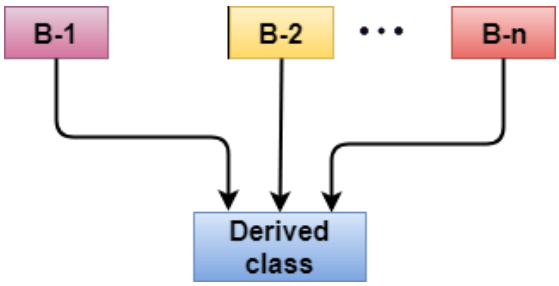

```
#include <bits/stdc++.h>
using namespace std;

class Employee {
public:
    int ID;
    float salary;
};

class Developer : public Employee {
public:
    float bonus;

    Developer(float salary, float bonus) {
        this->salary = salary;
        this->bonus = bonus;
    }
};

int main() {

    Developer dv1(20000, 1000);

    cout << "Salary: " << dv1.salary << endl
         << "Bonus: " << dv1.bonus << endl;

    return 0;
}
// output
// Salary: 20000
// Bonus: 1000
```

Multi Level Inheritance


```cpp
#include <bits/stdc++.h>
using namespace std;

class Boeing {
public:
    vector<string> materials;
    double payload;
    double fuel = 0;
    double velocity = 0;
    double acceleration;
    double wingspan;

    void set_velocity(double v) {
        this->velocity = v;
    }

    void fill_fuel(double intake) {
        this->fuel = intake;
    }
};

class Boeing_Commercial_Planes : public Boeing {
public:
    int passenger_capacity;
    double altitude;
    int parachutes = 20;
    bool auto_pilot = false;

    void set_altitude(double alt) {
        this->altitude = alt;
    }

    void turn_auto_pilot() {
        this->auto_pilot = !this->auto_pilot;
    }
};

class Boeing_747 : public Boeing_Commercial_Planes {
public:
    int engines;
```

```
    double range;

    void set_engines(int count) {
        this->engines = count;
    }

    void set_range(double r) {
        this->range = r;
    }
};

int main() {

    Boeing_747 New_Delhi_Chicago;
    New_Delhi_Chicago.fill_fuel(1000);
    New_Delhi_Chicago.wingspan = 21.34;
    New_Delhi_Chicago.set_altitude(20000);
    New_Delhi_Chicago.set_engines(4);

    cout << "Velocity: " << New_Delhi_Chicago.velocity << endl;
    cout << "Parachutes: " << New_Delhi_Chicago.parachutes << endl;

    return 0;
}

// Output
// Velocity: 0
// Parachutes: 20
```

Multiple Inheritance



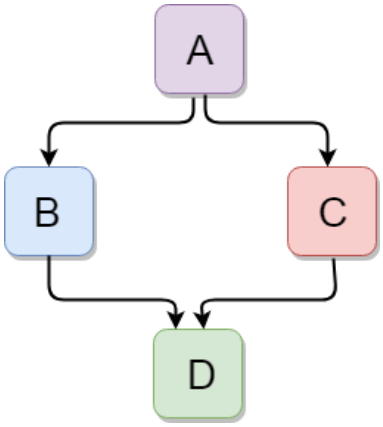

```cpp
#include <iostream>
using namespace std;

class Vehicle {
public:
    void move() {
        cout << "Vehicle is moving." << endl;
    }
};

class Engine {
public:
    void start() {
        cout << "Engine is starting." << endl;
    }
};

class Car : public Vehicle, public Engine {
public:
    void display() {
        move();
        start();
    }
};

int main() {
    Car car;
    car.display();
    return 0;
}

// output
// Vehicle is moving.
// Engine is starting.
```

Avoiding Ambuigity in name resolution

Sometimes in multiple inheritance different classes might be having functions & members with same names. This can lead to ambiguity of which function to call. To resolve this we use class-resolution operator hence stating which function or member to call of which class.

Hybrid Inheritance

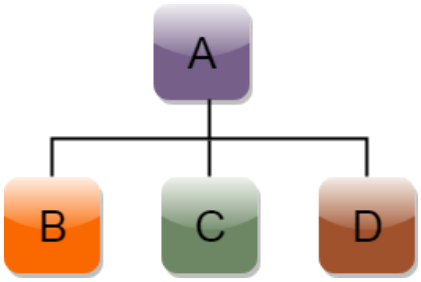
It's a combination of more than one type of inheritances.
For eg, here we combined multiple and multi-level inheritance.




```
#include <bits/stdc++.h>
using namespace std;

class Vehicle {
public:
    double mileage;
    int doors;
    string color;
    string fuel_type;
};

class Car : public Vehicle {
public:
    bool transmission;
    double speed;

    void accelerate(double acc, double time) {
        this->speed += (acc * time);
    }

    void breaking(double ret, double time) {
        this->speed -= (ret * time);
    }
};

class Truck : public Vehicle {
public:
    double capacity_tonnes;
    double ground_clearance;
    int height;
    int length;
    int width;
};

class Ford_F150 : public Car, public Truck {
public:
    string model_name;
    double max_torque;
```

```
    double rim_size;
    string breaking_type;

    Ford_F150() {
        this->model_name = "Ford F150";
        this->max_torque = 650;
        this->capacity_tonnes = 1.474;
        this->transmission = false;
        this->Truck::mileage = 8; // resolving ambuiguity arising as both Car
and Truck inherit Vehicle
        this->rim_size = 18;
    }

    void print() {
        // writing between "\033[1m \033[0m" makes text bold
        cout << "\033[1m Model: \033[0m" << this->model_name << endl;
        cout << "\033[1m Torque: \033[0m" << this->max_torque << " Nm" << endl;
        cout << "\033[1m Capacity: \033[0m" << this->capacity_tonnes * 1000 << "
kg" << endl;
        cout << "\033[1m Transmission: \033[0m" << (transmission == true ?
"Manual" : "Automatic") << endl;
        cout << "\033[1m Mileage: \033[0m" << this->Truck::mileage << " kmpl" <<
endl;
        cout << "\033[1m Rim Size: \033[0m" << this->rim_size << " inches" <<
endl;
    }
};

int main() {

    Ford_F150 ftruck1;
    ftruck1.print();

    return 0;
}

// output
// Model: Ford F150
// Torque: 650 Nm
// Capacity: 1474 kg
// Transmission: Automatic
// Mileage: 8 kmpl
// Rim Size: 18 inches
```

Heriarchial Inheritance




```
#include <iostream>
using namespace std;
class Shape {
public:
    int a;
    int b;
    void get_data(int n, int m) {
        a = n;
        b = m;
    }
};

class Rectangle : public Shape {
public:
    int rect_area() {
        int result = a * b;
        return result;
    }
};

class Triangle : public Shape {
public:
    int triangle_area() {
        float result = 0.5 * a * b;
        return result;
    }
};

int main() {

    Rectangle r;
    Triangle t;

    int length = 10, breadth = 10, base = 10, height = 10;

    r.get_data(length, breadth);
    int m = r.rect_area();
    cout << "Area of Rectangle: " << m << endl;

    t.get_data(base, height);
```

```
    float n = t.triangle_area();
    cout << "Area of Triangle: " << n << endl;
    return 0;
}

// Output
// Area of Rectangle: 100
// Area of Triangle: 50
```

Aggregation in C++

Aggregation is little bit related to inheritance. It simply means that one class depends on another class and they have "**has-a**" relationship between them.
For example, a car must have an engine.

```
#include <bits/stdc++.h>
using namespace std;

class Engine {
private:
    bool isRunning;
    double horsePower;
    string fuelType;

public:
    Engine(double hp, string fuel) : isRunning(false), horsePower(hp),
fuelType(fuel) {}

    void start() {
        if (isRunning) {
            cout << "Engine is already running." << endl;
            return;
        }
        isRunning = true;
        cout << "Engine started." << endl;
    }

    void stop() {
        if (!isRunning) {
            cout << "Engine is already stopped." << endl;
            return;
        }
        isRunning = false;
        cout << "Engine stopped." << endl;
    }

    double getHorsePower() {
        return horsePower;
    }

    string getFuelType() {
```

```
        return fuelType;
    }
};

class Car {
private:
    Engine engine;
    string modelName;
    int year;

public:
    Car(string model, int yr, double hp, string fuel) : engine(hp, fuel),
modelName(model), year(yr) {}

    void drive() {
        engine.start();
        cout << modelName << " is driving." << endl;
    }

    void stop() {
        engine.stop();
    }

    double getHorsePower() {
        return engine.getHorsePower();
    }

    string getFuelType() {
        return engine.getFuelType();
    }

    void printInfo() {
        cout << "Car Model: " << modelName << endl;
        cout << "Year: " << year << endl;
        cout << "Horse Power: " << engine.getHorsePower() << endl;
        cout << "Fuel Type: " << engine.getFuelType() << endl;
    }
};

int main() {
    Car c1("Ford Mustang", 2023, 460, "Gasoline");
    c1.printInfo();
    c1.drive();
```

```
    c1.stop();

    return 0;
}

// Output
// Car Model: Ford Mustang
// Year: 2023
// Horse Power: 460
// Fuel Type: Gasoline
// Engine started.
// Ford Mustang is driving.
// Engine stopped.
```

Encapsulation

Encapsulation is combining data and members (that manipulate them) under a single unit. Why?

- Data Hiding
    - Not letting others know about our data & code
- Abstraction
    - Hiding complexity of program from end user, only necessary feature to outside world
- Code reusability

For Eg, in a company we can divide different functionalities into departments of sections. Say, ‘sales’ and ‘accounts’ sections. Now, if an ‘accounts’ employee wants data of ‘sales’ he/she have to ask from the person handling ‘sales’. Thus we have achieved data hiding here and data reusability (no two copies between departments).

Features of Encapsulation

- We need an object to access data members and member functions of a class.
- C++ is doesn’t implement complete Encapsulation. For complete Encapsulation member functions should not be able to access data members & functions outside the class (in C++ we can access global data members and functions).
- Encapsulation helps in readability, and code maintenance.
- Encapsulation helps in controlling access level of data members.

```
#include <bits/stdc++.h>
using namespace std;

class bank {
private:
    int acc_no;
    string name;

public:
    bank(int number, string name) {
        this->acc_no = number;
        this->name = name;
    }

    void setAcc(int number) {
        this->acc_no = number;
    }

    void getAcc() {
        cout << this->acc_no << endl;
```

```
    }

    void setName(string name) {
        this->name = name;
    }

    void getName() {
        cout << this->name << endl;
    }
};

int main() {

    bank acc1(12345, "Tanmay, Sharma");
    acc1.getAcc();
    acc1.getName();

    acc1.setAcc(121);
    acc1.setName("John, Wick");

    acc1.getAcc();
    acc1.getName();

    return 0;
}

// Output
// 12345
// Tanmay, Sharma
// 121
// John, Wick
```

What do we do?

- Create class to combine data and functions
- Use access modifiers to control level of exposure

Abstraction

Abstraction is hiding of inner complexity from outside world, providing only required or necessary information only.
For eg, everyone knows how to drive a car but not everyone knows the inner mechanism.

Ways to abstract?

1. Using Classes
   Access modifiers for hiding

2. Using Header files
   No complete code is given, we only know about the utility. For eg, we know how to use pow() function but not know about complexity of math.h library

Types of Abstraction?

Data abstraction

- Hiding the representation and implementation
- Utilizing classes and objects for abstraction
- Eg, the above car example.

Control abstraction

- Hiding the logic of performing task
- Utilizing functions and header files
- Eg, you probably don't know how which algorithm does STL sort() function uses.

Advantages of Abstraction?

- Quite simple, you don't want others to see your code.
- You don't want to confuse your users by showing underlying complexities.
- You can change structure of code without affecting the outcome of it. Eg, you can change logical schema of database and the user probably won't know about it.
- It makes code more readable, you know where to work on code.

Using Classes & Objects

```cpp
#include <bits/stdc++.h>
using namespace std;

class area {

    // control the access of data members
    // such that they can be accesses only
    // inside the class
    // outside people can only utilize the
    // service
private:
    double radius = 0.0;
    double area = 0.0;

    // outside people can only
    // access the services available
    // to them
public:
    void setRadius(double radius) {
        this->radius = radius;
    }

    void getArea() {
        area = M_PI * radius * radius;
        cout << "Area of circle is: " << area << endl;
    }
};

int main() {

    area circle1;
    circle1.setRadius(10.1);
    circle1.getArea();

    return 0;
}

// Output
// Area of circle is: 320.474
```

Using Header files

Steps

- Create header file “myheader.h”
- Create cpp file for storing logic “myheader.cpp”
- Create main function somewhere else other than above files, import header and utilize

myheader.h

```
#ifndef MYHEADER_H
#define MYHEADER_H

const double PI = 3.14159265358979323846;
double calculateArea(double radius);

#endif
```

myheader.cpp

```
#include "myheader.h"

double calculateArea(double radius) {
    double area = PI * radius * radius;

    return area;
}
```

main.cpp

```
#include "myheader.h"
#include <iostream>

using namespace std;

int main() {

    double radius = 100.1;
    double area = calculateArea(radius);
    cout << "Area calculating from header files: " << area << endl;

    return 0;
}
```

Compile your code

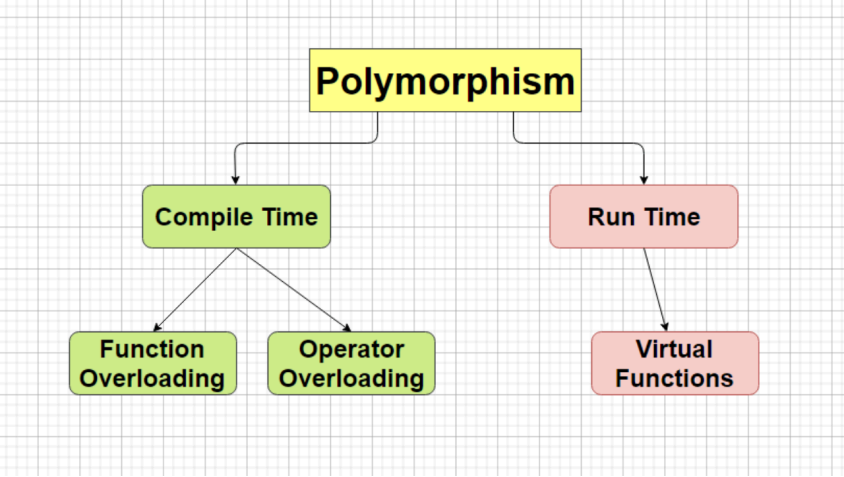
```
g++ -o main main.cpp myheader.cpp
```

If above command doesn't work, following might work:

- `g++ -c main.cpp`
- `g++ -c myheader.cpp`
- `g++ -o main main.o myheader.o`

Polymorphism

- Polymorphism means having different forms
- A man can be a father, son, brother at the same time
- Polymorphism is the ability to be available in more than one form



Compile time polymorphism

- It is called so because the compiler decides which method & operator functionality to use before compilation

Function Overloading

- There can exist multiple functions with same name but differentiated in number of parameters and type of parameters.
- It is called function overloading as we're utilizing the same function name for different purposes.
- It's said that function overloading only works when there's difference in number of parameters and type of parameters. Not based on return type. Why?
- It's so because function overloading is performed at compile time and compiler distinguished between these functions based on signatures.
- These signatures has nothing to do with return type. If signatures are different, then functions can be overloaded.

For eg,

```
int add(int a, int b) {
    return a + b;
}
```

In above code, the function signature is

```
add (int, int);
```

Function Overloading Example,

```
#include <bits/stdc++.h>
using namespace std;

class mathematics {
public:
    // function to add two variables
    int add(int a, int b) {
        return a + b;
    }

    // function to add three variables
    int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {

    mathematics m1;
    cout << "Call from two variable function: " << m1.add(5, 3) << endl;
    cout << "Call from three variable function: " << m1.add(8, 11, 11) << endl;

    return 0;
}

// output
// Call from two variable function: 8
// Call from three variable function: 30
```

Operator Overloading

- You've probably heard of that '+' is used as integer, float addition and as well as string concatenations.
- Yes, that's one big example used like billion times. We'll see how real operator overloading works.
- In general, if you've created a class of objects, you can't simply add the objects and wish to get a result. Well, that can be done. We can modify our operator to perform some task.

```cpp
#include <bits/stdc++.h>
using namespace std;

class coordinates {
public:
    int x;
    int y;

    coordinates(int inp_x, int inp_y) {
        this->x = inp_x;
        this->y = inp_y;
    }

    coordinates() {}; // default constructor

    // real operator overloading
    coordinates operator+(coordinates const &temp) {
        coordinates res;
        res.x = this->x + temp.x;
        res.y = this->y + temp.y;
        return res;
    }

    void print() {
        cout << "X: " << this->x << ", Y: " << this->y << endl;
    }
};

int main() {

    coordinates cd1(10, 12);
    coordinates cd2(11, 13);
```

```
    // we're overloading + here
    // and utilizing it to perform
    // addition for two objects
    coordinates cd3 = cd1 + cd2;
    cd3.print();

    return 0;
}
```

Functions that can’t be overloaded

- Functions that differ only by their return type.
- Static functions.
- Same parametric functions, say you passed a pointer in one function and an array in another (since array names acts as pointer to first element).
- Parameters that are different only by a const. Eg, passing *int* and *const int* is going to be considered same.

Run-Time Polymorphism

- It is called so because the decision of selection which function to call is taken at runtime.
- It is achieve using virtual functions.
- Why use virtual functions?
    - If we don’t use virtual functions compiler simply replaces the implementation of base class function with child class function.
    - And not using virtual functions is called compile-time overriding.
- In run-time polymorphism, function signature remains same.
- In later languages like Java, all functions are virtual by default.
- Compiler manages pointers for referencing to the required virtual function.

```cpp
#include <bits/stdc++.h>
using namespace std;

class Animal {
public:
    // we didn't made virtual function
    // and hence no child function will be overridden
    void makeSound() {
        cout << "Animal made sound!" << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() {
        cout << "Dog barks!" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() {
        cout << "Cat meows!" << endl;
    }
};

class Lion : public Animal {
public:
    void makeSound() {
        cout << "Lion Roars!" << endl;
```

```
    }
};

int main() {

    Animal *animals[3] = {new Dog(), new Cat(), new Lion()};

    for (auto sound : animals) {
        sound->makeSound();
    }

    return 0;
}

// Output
// Animal made sound!
// Animal made sound!
// Animal made sound!
```

But in following code we'll use virtual functions to call the targeted *makeSound()* function.

```
#include <bits/stdc++.h>
using namespace std;

class Animal {
public:
    // defining a virtual function
    virtual void makeSound() {
        cout << "Animal made sound!" << endl;
    }
};

class Dog : public Animal {
public:
    // overriding makeSound()
    void makeSound() override {
        cout << "Dog barks!" << endl;
    }
};

class Cat : public Animal {
public:
    // overriding makeSound()
    void makeSound() override {
        cout << "Cat meows!" << endl;
    }
};

class Lion : public Animal {
public:
    // overriding makeSound()
    void makeSound() override {
        cout << "Lion Roars!" << endl;
    }
};

int main() {

    Animal *animals[3] = {new Dog(), new Cat(), new Lion()};

    for (auto sound : animals) {
```

```
        sound->makeSound();
    }

    return 0;
}

// Output
// Dog barks!
// Cat meows!
// Lion Roars!
```

Advantages and Disadvantages of Run-Time function overriding

Advantages	**Disadvantages**
Functions to implement are selected at runtime	Errors can be detected only at run-time (if any)
Common functions can be reused as per future requirements	Virtual function calls are quite slower than normal function calls
Changes made in base class directly applies those in child class	These procedures are dynamic in nature and add an extra layer of overheads

Function Overloading vs Overriding

Function Overloading	**Function Overriding**
Scope is generally same for two overloaded functions	Scope in Overriding is different
Function signature changes	Function signature remains same
This can be executed without the need for inheritance from parent class	This cannot be executed without inheriting from parent class
It is resolved at compile time	It comprises of both Compile-Time and Run-TIme overriding

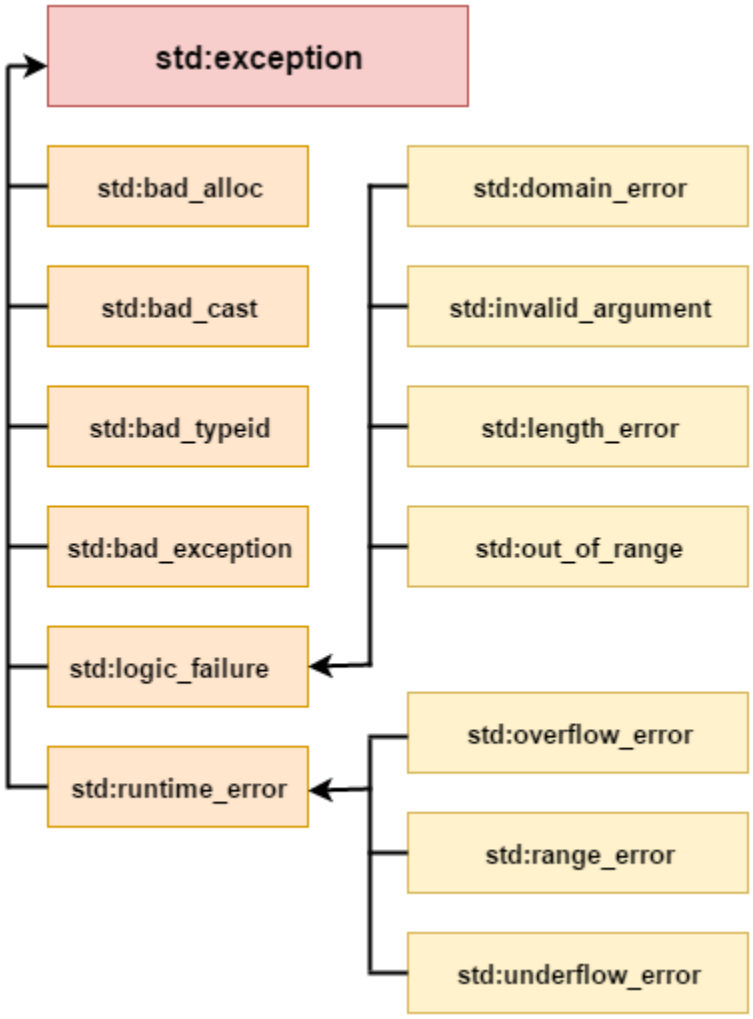
Access parent function from child object,

```
Animal *animals[3] = {new Dog(), new Cat(), new Lion()};


for (auto sound : animals) {
    sound->Animal::makeSound();
    // sound->makeSound();
}
```

Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from *std::exception* class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.



std::exception
Base class for all standard C++ exceptions.
Provides a virtual member function what() that returns a descriptive string about the exception.

std::bad_alloc
Thrown when memory allocation fails using new.

std::bad_cast
Thrown when a type cast operation fails.

std::bad_typeid
Thrown when a type identification operation fails.

std::bad_exception
Thrown when an exception is caught that is not a descendant of std::exception.

std::domain_error
Thrown when a function argument is outside the domain of its definition.

std::invalid_argument
Thrown when a function argument is invalid.

std::length_error
Thrown when a string or container is too long.

std::out_of_range
Thrown when an index or iterator is out of range.

std::logic_error
Thrown when a logical error occurs during program execution.

std::overflow_error
Thrown when an arithmetic operation results in an overflow.

std::range_error
Thrown when a mathematical operation results in a value outside the range of representable numbers.

std::underflow_error
Thrown when an arithmetic operation results in an underflow.

std::runtime_error
Thrown when a runtime error occurs during program execution.

In C++, we use 3 keywords to perform exception handling

try Block

- Encloses a block of code that might throw an exception.
- If an exception is thrown within the try block, the program control immediately jumps to the appropriate catch block.

catch Block

- Handles exceptions that are thrown within the corresponding try block.
- Can specify multiple catch blocks to handle different types of exceptions.
- The catch block's parameter type must match or be a base class of the exception type thrown.

throw Statement

- Used to explicitly throw an exception.
- Takes an object of a class derived from std::exception as an argument.
- When an exception is thrown, the program control jumps to the first matching catch block.

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

int main() {
    int num1, num2;

    try {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;

        if (num2 == 0) {
            throw runtime_error("Division by zero is not allowed.");
        }

        int result = num1 / num2;
        cout << "Result: " << result << endl;
    } catch (runtime_error &e) {
        cerr << "Error: " << e.what() << endl;
```

```
    }

    return 0;
}

// Output
// Enter two numbers: 12 0
// Error: Division by zero is not allowed.
```

Custom Exceptions

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

class MyException : public exception {
public:
    const char *what() const noexcept override {
        return "This is a custom exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch (MyException &e) {
        cerr << "Caught custom exception: " << e.what() << endl;
    }

    return 0;
}

// Output
// Caught custom exception: This is a custom exception
```

Files and Streams

In C++ programming we are using the ***iostream*** standard library, it provides cin and cout methods for reading from input and writing to output respectively.

To read and write from a file we are using the standard C++ library called ***fstream***.

fstream Library
Provides classes for input, output, and input/output operations on files.
Includes the following classes:

- ifstream: Input file stream (reading from a file).
- ofstream: Output file stream (writing to a file).

Use the open() member function to open a file.
Specify the file name and mode (e.g., ios::in, ios::out, ios::app).

// append text to file

```
#include <bits/stdc++.h>
using namespace std;

int main() {

    fstream filestream;

    filestream.open("file.txt", ios::app);

    if (filestream.is_open()) {
        filestream << endl
                   << "Hello, Tanmay Sharma!" << endl;

        filestream.close();
    }

    return 0;
}
```

// reading from file

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {

    fstream file;

    file.open("file.txt", ios::in);

    // one way, reading character by character
    if (file.is_open()) {
        char ch;
        while (file.get(ch)) {
            cout << ch;
        }
        file.close();
    }

    cout << endl;

    // second way, reading line by line
    file.open("file.txt", ios::in);
    if (file.is_open()) {
        string str;
        while (getline(file, str)) {
            cout << str << endl;
        }
        file.close();
    }

    return 0;
}
```

// writing to file

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {

    fstream file;
    file.open("file.txt", ios::out);

    // putting a character
    if (file.is_open()) {
        file.put('A');
        file.close();
    }

    file.open("file.txt", ios::out);

    // putting a line
    if (file.is_open()) {
        file << "Ghostbusters!!" << endl;
        file << 66 << endl;
        file.close();
    }

    return 0;
}
```

Feature	Reading	Writing	Appending
Mode	`ios::in`	`ios::out`	`ios::app`
Operation	Reads from the file	Writes to file, erases pre-existing data	Starts adding data just after the pointer's position
File Creation	File must exist	If file is not present, it creates a new one	Appends to existing and if not present then creates one

Multi-threading in C++

Multithreading in C++ is a technique that allows a single program to execute multiple tasks concurrently within a single process. This can improve performance and responsiveness, especially for applications that need to handle multiple tasks simultaneously.

Key Concepts

1. Thread: A thread is a lightweight process that shares the same memory space with other threads within the same process.
2. Process: A process is a program that runs independently and has its own memory space.
3. Concurrency: Multiple tasks are executed seemingly simultaneously, but they may not be truly parallel due to time-sharing.
4. Parallelism: Multiple tasks are executed simultaneously on multiple cores or processors.

Benefits of Multithreading

1. Improved performance: Can speed up applications by utilizing multiple cores or processors.
2. Responsiveness: Can make applications more responsive by allowing tasks to be executed in parallel.
3. Modularity: Can break down complex tasks into smaller, more manageable threads.

Challenges of Multithreading

1. Synchronization: Ensuring that threads access shared resources in a safe and consistent manner.
2. Deadlock: A situation where two or more threads are waiting for each other to release resources, resulting in a standstill.
3. Race conditions: When multiple threads access shared data simultaneously, leading to unpredictable results.

```
#include <iostream>
#include <thread>
#include <unistd.h>
using namespace std;

void taskA() {
    for (int i = 0; i < 10; i++) {
        sleep(1);
        cout << "taskA: " << i << endl;
        fflush(stdout);
    }
}
```

```
void taskB() {
    for (int i = 10; i < 20; i++) {
        sleep(1);
        cout << "taskB: " << i << endl;
        fflush(stdout);
    }
}

int main() {

    thread t1(taskA);
    thread t2(taskB);

    t1.join();
    t2.join();
    return 0;
}

// Output
```

taskB: 10
taskA: 0
taskB: 11
taskA: 1
taskA: 2taskB: 12

taskA: 3
taskB: 13
taskA: 4
taskB: 14
taskA: 5
taskB: 15
taskA: 6
taskB: 16
taskA: 7
taskB: 17
taskA: 8
taskB: 18
taskA: 9
taskB: 19