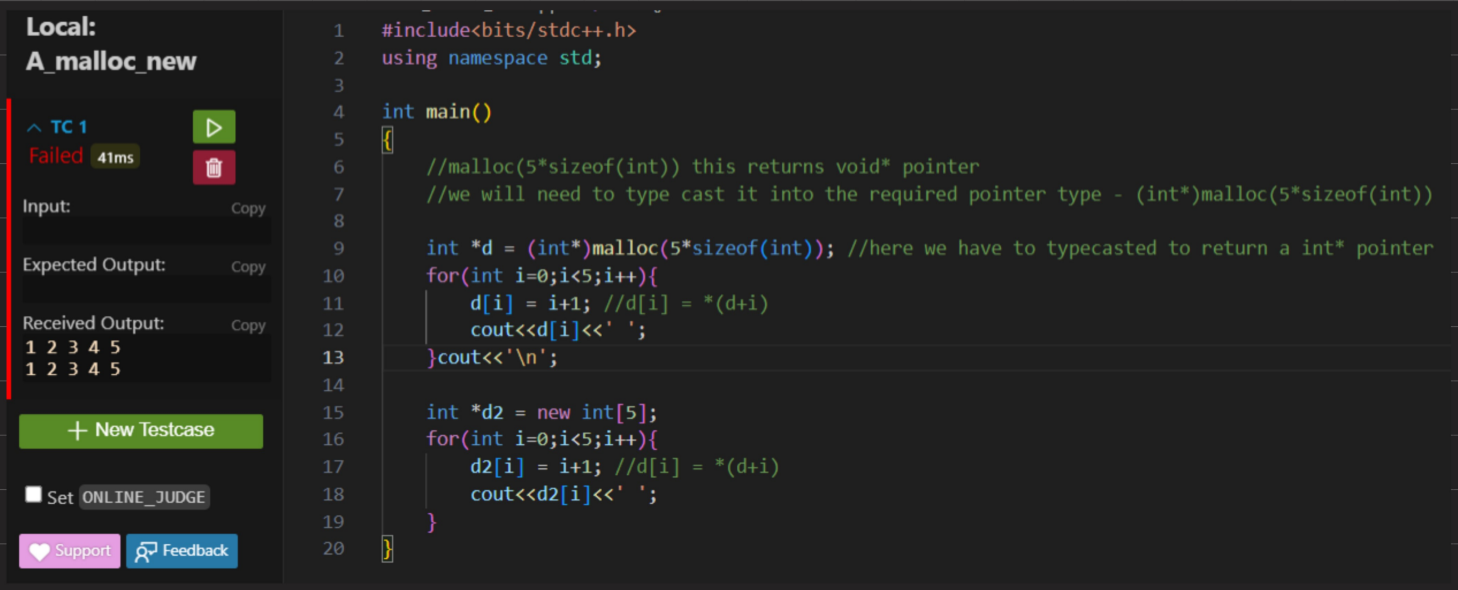
Malloc vs new :



Aspect	`malloc` in C	`new` in C++
Syntax	`void* ptr = malloc(size_in_bytes);`	`Type* ptr = new Type[size];`
Initialization	Does not initialize the allocated memory (contains garbage)	Initializes the allocated memory; calls constructors
Type Safety	Requires explicit type casting from `void*`	Returns a pointer of the correct type without casting
Error Handling	Returns `NULL` on failure (requires manual check)	Throws `std::bad_alloc` exception on failure (catchable)
Deallocation	Memory freed using `free(ptr);`	Memory freed using `delete ptr;` or `delete[] ptr;` for arrays

OOPs

What is object ?

Ans : object → entity → state / properties
                      → behaviour

ex: Game → Hero → Name (Paul)
(Takken)   → Health (70 /.)
                  → level (10, 12, ..)
                  } properties

Hero → behaviour
  → attack()
  → defense()
  → dance()

Class → user - defined datatype.

int a;
string s;
char c;
} inbuit

Hero Shahil;
class

Object is an instance of Class.

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

class Hero{ //Hero is the class name
    //properties
    char name[100]; //100 byte
    int health; //4 byte
    int level;  //4 byte
};

int main()
{
    Hero h1; //h1 is an object of class hero
    cout<<sizeof(h1)<<'\n';
    return 0;
}
```

class : Hero
Object : h1

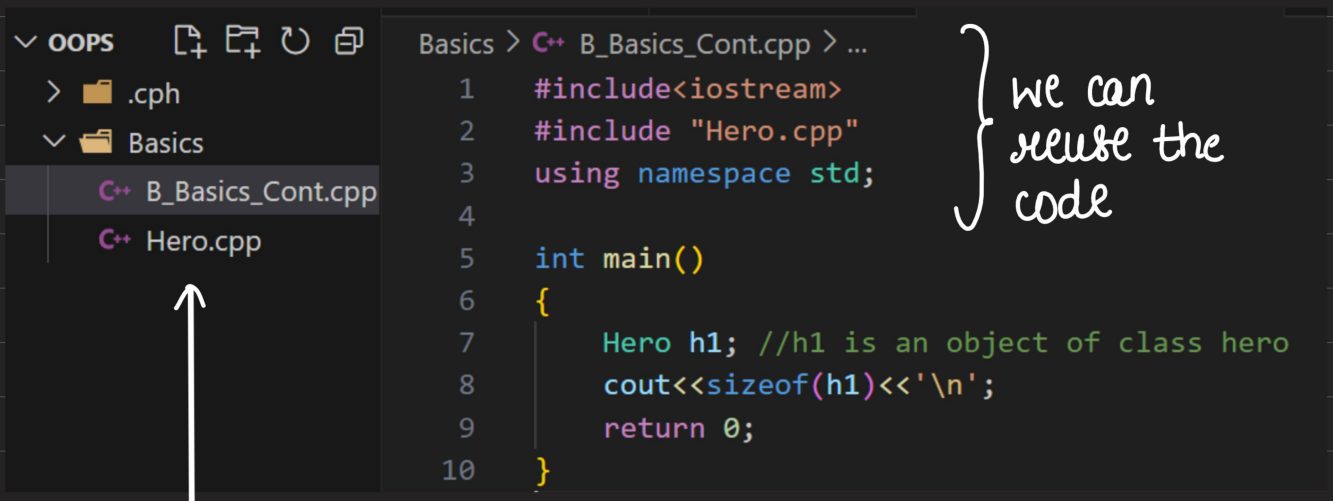
Received Output: Copy
108

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

class Empty_Class{
    //empty class has size 1
    //this is to keep track of its identification
};

int main()
{
    Empty_Class h1; //h1 is an object
    cout<<sizeof(h1)<<'\n';
    return 0;
}
```

Received Output: Copy

```
1
```

empty class has size = 1



we can reuse the code

Note: name of file should be same as name of class.

Now,

We can access properties (which are public) using '.' dot operator.

Access modifiers

- Public
- Private
- Protected

[By default, its private]

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

class Hero
{
private:
    int level;  //4 byte
public:
    char name[100]; //100 byte
    int health; //4 byte

    void print_health() //this function is public
    {
        cout<<health<<'\n'; //health can be accesed only within the class
    }
};

int main()
{
    Hero h1; //h1 is an object of class hero
    cout<<"Health: "<<h1.health<<'\n';
    cout<<"Level: "<<h1.level<<'\n'; //error due to private member
    h1.print_health();
}
```

Since level is private, we can access it inside class only.

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

class Hero
{
public:
    char name[100]; //100 byte
    int health; //4 byte
    int level;  //4 byte

    void print_health() //this function is public
    {
        cout<<health<<'\n'; //health can be accesed only within the class
    }
};

int main()
{
    Hero h1; //h1 is an object of class hero
    h1.health = 70;
    h1.level = 10;
    cout<<"Health: "<<h1.health<<'\n';
    cout<<"Level: "<<h1.level<<'\n'; //error due to private member
    h1.print_health();
}
```

once made public we can access properties using '.' dot operator.

Getter and Setter :

function that helps get/set value of private properties.

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

class Hero
{
private:
    int health; //4 byte
public:
    char name[100]; //100 byte
    int level;  //4 byte

    int get_Health(){
        return health;
    }
    void set_Health(int h){
        health = h;
        return;
    }

};

int main()
{
    Hero h1;
    h1.set_Health(70);
    cout<<"Health: "<<h1.get_Health()<<'\n'; return 0;
}
```

Received Output: Copy
Health: 70

getter

setter

this is static allocation

Declaration of object

hero h1;

```
class Hero{
 int health;
 char level;
}
```

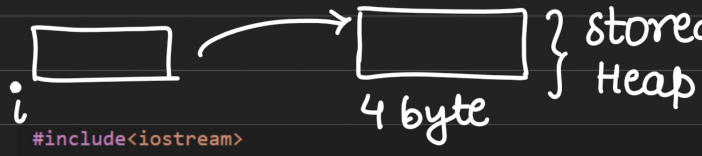
sizeof (h1) = 8

Why not 5 ?

Ans: padding / memory alignment

Dynamic allocation

int * i = new int ;



Similarly

Hero *h = new Hero;

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

class Hero
{
private:
    int health; //4 byte
public:
    char name[100]; //100 byte
    int level;  //4 byte
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
};

int main()
{
    Hero* h = new Hero;
    h->set_Health(70); //same as (*h).set_Health(70)
    cout<<"Health: "<<h->get_Health()<<'\n';
}
```

Received Output:
Health: 70

The size of the class `class{int a; char c;};` is 8 instead of 5 due to **memory alignment** and **padding**.

1. **Memory Alignment**:
    - Modern processors are optimized for accessing data at specific memory boundaries. For instance, an `int` is usually aligned on a 4-byte boundary, meaning its memory address should be a multiple of 4. Similarly, different data types have different alignment requirements.
2. **Padding**:
    - To satisfy the alignment requirements, the compiler may add padding bytes between members of the class or structure. This ensures that each member is correctly aligned in memory.

Without padding, the layout would be:

```scss
| a (4 bytes) | c (1 byte) |
```

However, for alignment purposes, the compiler will add padding after the `char` to ensure the size of the class is a multiple of the largest alignment requirement (which is 4 bytes in this case).

Thus, the actual layout in memory will be:

```scss
| a (4 bytes) | c (1 byte) | padding (3 bytes) |
```

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

class Hero
{
private:
    int health; //4 byte
public:
    char name[100]; //100 byte
    int level;  //4 byte
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
};

int main()
{
    Hero* h = new Hero; //h is a pointer to the address
    cout<<sizeof(h)<<'\n'; //this size is 4
    cout<<sizeof(*h)<<'\n';//this size is 108

    h->set_Health(70); //same as (*h).set_Health(70)
    cout<<"Health: "<<h->get_Health()<<'\n';
}
```

Received Output: Copy

```
4
108
Health: 70
```

Constructor

Whenever object is created a constructor is called. Constructor invoke during object creation with no return type.

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

class Hero
{
private:
    int health; //4 byte
public:
    char name[100]; //100 byte
    int level;  //4 byte

    int get_Health(){
        return health;
    }
    void set_Health(int h){
        health = h;
        return;
    }

};

int main()
{
    Hero h1;
    h1.set_Health(70);
    cout<<"Health: "<<h1.get_Health()<<'\n'; return 0;
}
```

Received Output: Copy

```
Health: 70
```

Constructor

When h1 is created default constructor is called.

h1.Hero();

Default constructor has no i/p parameter

Local: Hero

TC 1
Failed 22ms

Input: Copy

Expected Output: Copy

Received Output: Copy
```
Constructor Called
Constructor Called
Constructor Called
```

+ New Testcase

Set ONLINE_JUDGE

Support Feedback

Basics > Hero.cpp > main()

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

class Hero
{
private:
    int health;
public:
    char name[100];
    int level;
    Hero(){ //changing the default constructor
        cout<<"Constructor Called\n";
    }
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
};

int main()
{
    Hero h;
    Hero *h1 = new Hero;
    Hero *h2 = new Hero();
    return 0;
}
```

During declaration constructor is called.

We have overwriteen the default constructor.

Parameterized constructor

Local: Hero

TC 1
Failed 25ms

Input: Copy

Expected Output: Copy

Received Output: Copy
```
Shahil 1 100
Shahil 1 100
```

+ New Testcase

Set ONLINE_JUDGE

Support Feedback

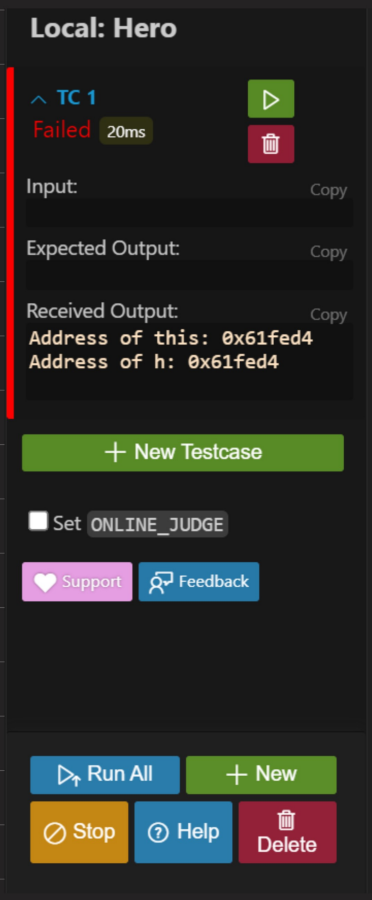
Run All + New

Stop Help Delete

```cpp
class Hero
{
private:
    int health;
public:
    string name;
    int level;
    Hero(){
        cout<<"Default Constructer\n";
    }
    Hero(int health, int level, string name){ //Parameterized constructor
        //parameters has same name as properties, so, here we use this keyword
        this->health = health;
        this->level = level;
        this->name = name;
    }
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
};

int main()
{
    Hero h = Hero(100,1,"Shahil");
    cout<<h.name<<' '<<h.level<<' '<<h.get_Health()<<'\n';
    Hero *h1 = new Hero(100,1,"Shahil");
    cout<<h1->name<<' '<<h1->level<<' '<<h1->get_Health()<<'\n';
    return 0;
}
```

↳ it refers to health with closest scope.

} using parameter. constructor




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

class Hero
{
private:
    int health;
public:
    string name;
    int level;
    Hero(){
        cout<<"Default Constructer\n";
    }
    Hero(int health, int level, string name){ //Parameterized constructor
        //parameters has same name as properties, so, here we use this keyword
        cout<<"Address of this: "<<this<<'\n';
        this->health = health;
        this->level = level;
        this->name = name;
    }
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
};

int main()
{
    Hero h = Hero(100,1,"Shahil");
    cout<<"Address of h: "<<&h<<'\n';
    return 0;
}
```

Both 'h' & 'this' points to same address.
'this' actually stores the address of current object.

Constructor's Chod

Even if you create one constructor (with or without parameter(s)), default constructor gets deleted.

```cpp
class Hero
{
private:
    int health;
public:
    string name;
    int level;
    Hero(int health, int level, string name){ //Parameterized constructor
        //parameters has same name as properties, so, here we use this keyword
        this->health = health;
        this->level = level;
        this->name = name;
    }
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
};

int main()
{
    Hero h;
    return 0;
}
```

} error as default constructor is deleted.

Local: Hero

TC 1 Failed 23ms

Input: Copy

Expected Output: Copy

Received Output: Copy
Name: Shahil Health: 100 Level: 1
Name: Shahil Health: 100 Level: 1

+ New Testcase

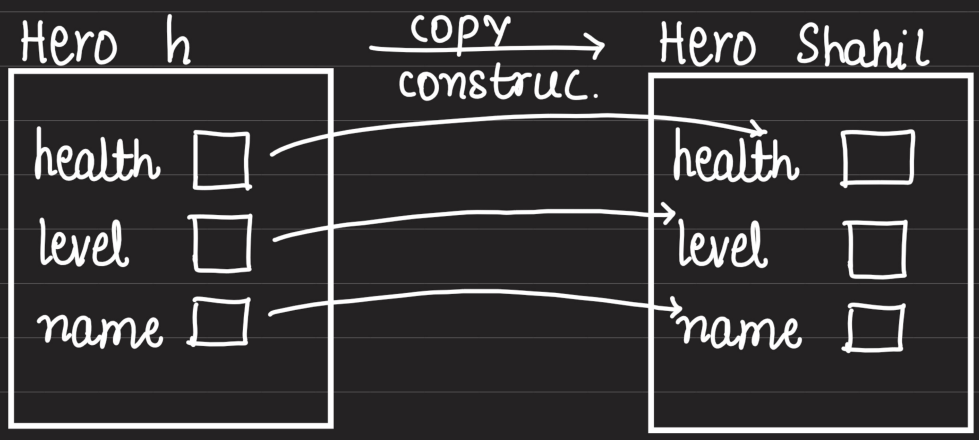
Set ONLINE_JUDGE

Support Feedback

```cpp
class Hero
{
private:
    int health;
public:
    string name;
    int level;
    Hero(){
        cout<<"Default Constructer\n";
    }
    Hero(int health, int level, string name){ //Parameterized constructor
        //parameters has same name as properties, so, here we use this keyword
        this->health = health;
        this->level = level;
        this->name = name;
    }
    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
    void print(){
        cout<<"Name: "<<this->name<<" Health: "<<this->health<<" Level: "<<this->level<<'\n';
    }
};

int main()
{
    Hero h(100, 1, "Shahil");
    h.print();
    Hero Shahil(h);
    Shahil.print();
    return 0;
}
```

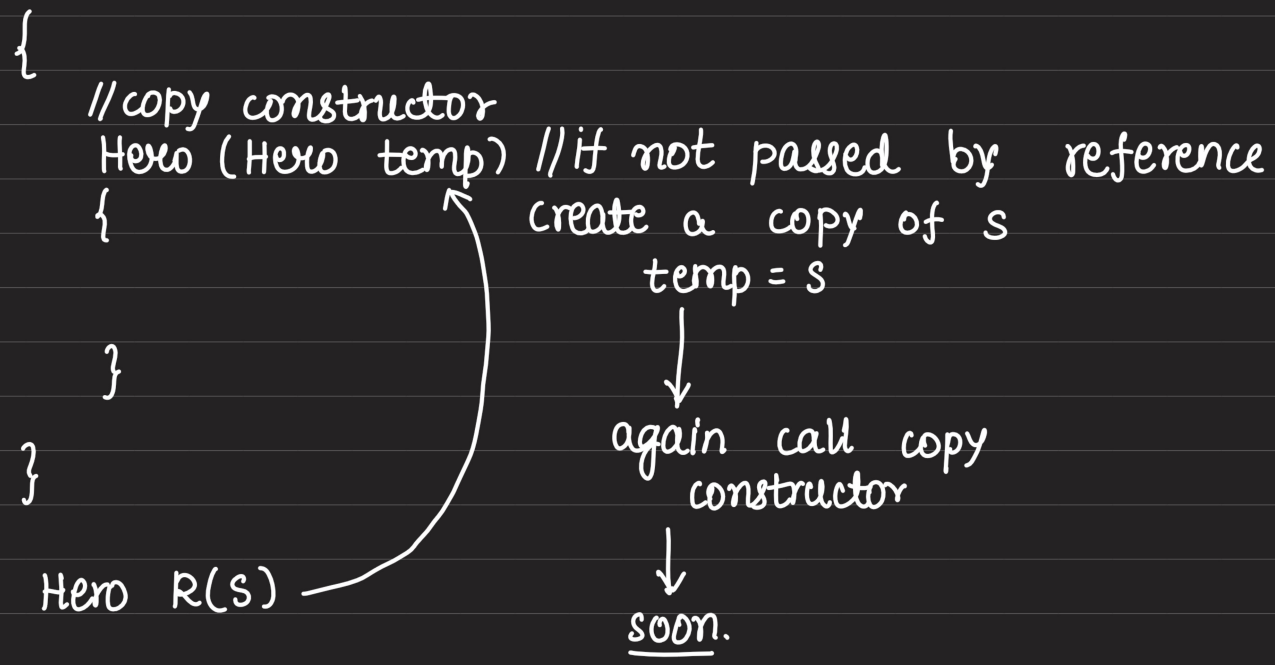
} copy constructor

Copy constructor is generated by itself




```cpp
class Hero
{
private:
    int health;
public:
    string name;
    int level;
    Hero(){
        cout<<"Default Constructer\n";
    }
    Hero(int health, int level, string name){ //Parameterized constructor
        //parameters has same name as properties, so, here we use this keyword
        this->health = health;
        this->level = level;
        this->name = name;
    }
    //copy constructor
    Hero(Hero &temp){ //we will have to pass by reference only
        this->health = temp.health;
        this->level = temp.level;
        this->name = temp.name;
    }

    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }
    void print(){
        cout<<"Name: "<<this->name<<" Health: "<<this->health<<" Level: "<<this->level<<'\n';
    }
};
```

} if not passed by reference, then a copy of temp will be passed into copy constructor. But for copying, we again need to copy it, this is infinite loop.



Shallow and Deep copy

default copy constructor → shallow copy.

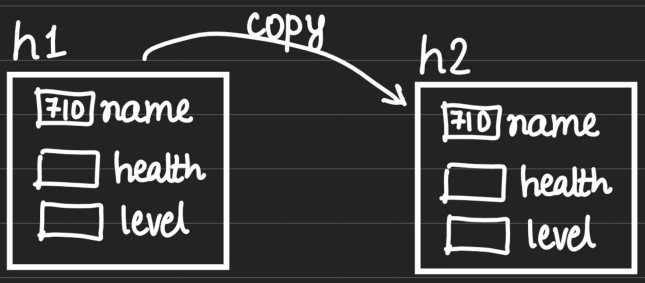
```cpp
class Hero
{
private:
    int health;
public:
    char *name;
    int level;
    Hero(){
        cout<<"Default Constructer\n";
        name = new char[20];
    }
    Hero(int health, int level, char name[]){
        this->health = health;
        this->level = level;
        strcpy(this->name,name);
    }

    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }

    string getName(){ return this->name; }
    void set_name(char* name){this->name = name; return; }

    void print(){
        cout<<"Name: "<<this->name<<" Health: "<<this->health<<" Level: "<<this->level<<'\n';
    }
};
```

```cpp
int main()
{
    Hero h1;
    h1.set_Health(100);
    char name[7] = "Shahil";
    h1.set_name(name);
    h1.level = 1;
    h1.print();

    //using default copy constructor ->makes a shallow copy
    Hero h2(h1);
    h2.print();

    h1.name[0] = 'G';
    //After changing h1
    h1.print();
    h2.print();
    return 0;
}
```

→ we only changed h1.name[0], but it got changed in h2 as well.
Actually name is a char* (char[])

Received Output: Copy

```
Default Constructer
Name: Shahil Health: 100 Level: 1
Name: Shahil Health: 100 Level: 1
Name: Ghahil Health: 100 Level: 1
Name: Ghahil Health: 100 Level: 1
```



so basically, in shallow copy, we access same memory
In deep copy, we will create a new address & initialize its value to 'Shahil' and then copy the start pointer.

How to do deep copy ?

```
class Hero
{
private:
    int health;
public:
    char *name;
    int level;
    Hero(){
        cout<<"Default Constructer\n";
        name = new char[20];
    }
    Hero(int health, int level, char name[]){
        this->health = health;
        this->level = level;
        strcpy(this->name,name);
    }
    //user defined copy constructor
    Hero(Hero &temp){
        char *ch = new char[strlen(temp.name)+1]; //creating a hard copy
        strcpy(ch,temp.name);
        this->name = ch;
        this->level = temp.level;
        this->health = temp.health;
    }

    int get_Health(){ return health; }
    void set_Health(int h){ health = h; return; }

    string getName(){ return this->name; }
    void set_name(char* name){this->name = name; return; }

    void print(){
        cout<<"Name: "<<this->name<<" Health: "<<this->health<<" Level: "<<this->level<<'\n';
    }
};
```

```
int main()
{
    Hero h1;
    h1.set_Health(100);
    char name[7] = "Shahil";
    h1.set_name(name);
    h1.level = 1;
    h1.print();

    //using default copy constructor ->makes a shallow copy
    Hero h2(h1);
    h2.print();

    h1.name[0] = 'G';
    //After changing h1
    h1.print();
    h2.print();
    return 0;
}
```

Received Output: Copy

```
Default Constructer
Name: Shahil Health: 100 Level: 1
Name: Shahil Health: 100 Level: 1
Name: Ghahil Health: 100 Level: 1
Name: Shahil Health: 100 Level: 1
```

Copy assignment operator

Hero a (10, "c")
Hero b (20, "b")

a = b } ⟶ a.level = b.level / a.name = b.name.

So, addresses will be copied directly. So, this is also shallow copy.

```cpp
int main()
{
    Hero h1, h2;
    char name1[7] = "Shahil", name2[7] = "Jhahil";
    h1.set_name(name1); h2.set_name(name2);
    h1.level = 1; h2.level = 1;
    h1.set_Health(100); h2.set_Health(100);

    h1.print(); h2.print(); cout<<'\n';

    h2 = h1; //h2 and h1 name* addresses are now same
    h1.print(); h2.print(); cout<<'\n';
    h1.name[0] = 'G';
    //After changing h1
    h1.print(); h2.print();
    return 0;
}
```

Received Output: Copy

```
Default Constructer
Default Constructer
Name: Shahil Health: 100 Level: 1
Name: Jhahil Health: 100 Level: 1

Name: Shahil Health: 100 Level: 1
Name: Shahil Health: 100 Level: 1

Name: Ghahil Health: 100 Level: 1
Name: Ghahil Health: 100 Level: 1
```

Destructor : to deallocate memory.

Similarly, it gets automatically created during creation of an object.

It is also named same as Class name. It has no parameters & no return type.

How to differentiate b/w constructor and destructor?
Ans : ~ tilda.

```cpp
~Hero(){
    cout<<"Destructor Call\n";
}
```

```cpp
int main()
{
    Hero h1, h2;
    Hero *h3 = new Hero();
    char name1[7] = "Shahil", name2[7] = "Jhahil";
    h1.set_name(name1); h2.set_name(name2);
    h1.level = 1; h2.level = 1;
    h1.set_Health(100); h2.set_Health(100);
    cout<<'\n'; h1.print(); h2.print(); cout<<'\n';
    return 0;
}
```

Received Output:

```
Default Constructer
Default Constructer
Default Constructer

Name: Shahil Health: 100 Level: 1
Name: Jhahil Health: 100 Level: 1

Destructor Call
Destructor Call
```

for statically created object, destructor is automatically called

for dynamically created object, we need to manually call.
i.e, `delete(h3);`

Static Keyword

```
class Hero
{
private:
    int health;
public:
    char *name;
    int level;
    static int time_limit_for_game; //this is independent for all
    Hero(){
        cout<<"Default Constructer\n";
        name = new char[20];
    }
    Hero(int health, int level, char name[]){
        this->health = health;
        this->level = level;
        strcpy(this->name,name);
    }
    void print(){
        cout<<"Name: "<<this->name<<" Health: "<<this->health<<" Level: "<<this->level<<'\n';
    }
};

int Hero::time_limit_for_game = 5; //initilizing (:: - this is scope resolution operator)
```

```
int main()
{
    cout<<Hero::time_limit_for_game<<'\n';
    return 0;
}
```

Received Output:

```
5
```

Static variables can be used without declaring an object. As it is same for all objects, so, it belongs to the class, not to a particular object.

It is recommended to access it using :: , scope resolution operator.

- static functions:
  - ↳ can be accessed directly without creating object
  - ↳ they don't have this keyword as they donot belong to a particular object.
  - ↳ they can only access static members.

```cpp
class Hero
{
private:
    int health;
public:
    char *name;
    int level;
    static int time_limit_for_game; //this is independent for all
    Hero(){
        cout<<"Default Constructer\n";
        name = new char[20];
    }
    Hero(int health, int level, char name[]){
        this->health = health;
        this->level = level;
        strcpy(this->name,name);
    }
    static int random(int p){
        time_limit_for_game = p;
        return time_limit_for_game;
    }
    void print(){
        cout<<"Name: "<<this->name<<" Health: "<<this->health<<" Level: "<<this->level<<'\n';
    }
};

int Hero::time_limit_for_game = 5; //initilizing (:: - this is scope resolution operator)
```

```cpp
int main()
{
    cout<<Hero::time_limit_for_game<<'\n';
    cout<<Hero::random(100)<<'\n';
    return 0;
}
```

Received Output:

```
5
100
```

Summary

Class / Object → data members / Behaviour/ functions | Padding / Memory aligment

Access modifiers : public / private / protected

Allocation : Static / Dynamic

Construction: Default / Simple / Parameterized / Copy

Deep copy / Shallow copy

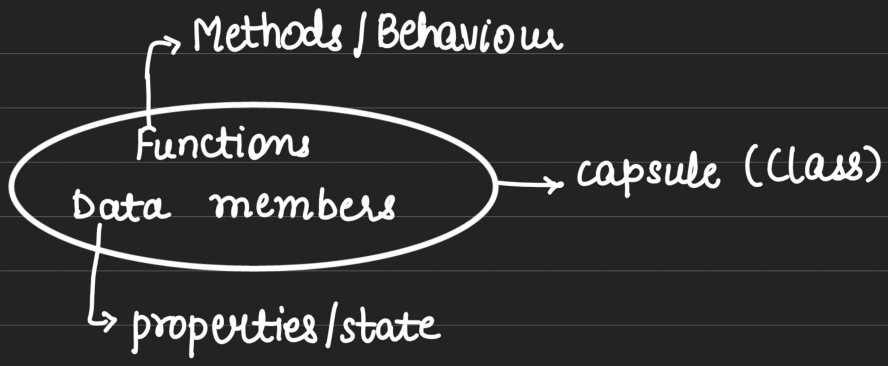
Destructor

Static : data members / functions.

Pillar of OOPs

→ Encapsulation, Inheritance
→ Polymorphism, Abstraction

1. Encapsulation : wrapping up data members and functions



NOTE: Fully encapsulated class : all data members are private
(D.M can be used only inside the class)

Advantage :

1. We can hide data members → security
2. We can make our class read only (only getter, no setter)
3. Code Reusability
4. Encapsulation → ease in unit testing

TC 1
Failed 21ms

Input: Copy

Expected Output: Copy

Received Output: Copy
5
32

+ New Testcase

Set ONLINE_JUDGE

Support

Feedback

```
4   class student {
5   private:
6       string name; //24 bytes
7       int age; //4 byte
8       int height; //4 byte
9   public:
10      student(){
11          name = "";
12          age = 5;
13          height = 50; //cm
14      }
15      int getAge(){ return this->age; }
16  };
17
18  int main()
19  {
20      student sgv;
21      cout<<sgv.getAge()<<'\n';
22      cout<<sizeof(sgv)<<'\n';
23      return 0;
24  }
```

Inheritance

↳ Qualities inherited from parents

class A → fun1 fun2

↓ ↓ inherited

class B fun1 fun2

eg: class human → parent class / super class / Base class

↳ height
↳ weight
↳ age

class Male → inherit

class female → inherit → Sub class / Child class

1. Public mode

class Child_name : mode Parent_name

```
class Human{
public:
    int height;
    int weight;
    int age;

    int getAge(){ return this->age; }
    int setweight(int w){ this->weight = w; }

};
class Male: public Human {
public:
    string color;
    void sleep(){
        cout<<"Male is Sleeping"<<'\n';
    }
};

int main()
{
    Male obj1;
    obj1.
    retur
}
```

```
age
color
getAge
height
setweight
sleep
weight
```

mode
- Public
- Private
- protected

Private members cannot be inherited in any mode.

We inherited properties & functions of Human class. Along with that we have extra features.

```cpp
class Human{
public:
    int height;
    int weight;
    int age;
    Human(){
        height = weight = age = 0;
    }
    int getAge(){ return this->age; }
    int setweight(int w){ this->weight = w; }

};
class Male: public Human {
public:
    string color;
    Male(){
        height = 10;
        color = "Black";
    }
    void sleep(){
        cout<<"Male is Sleeping"<<'\n';
    }
};
```

base / parent / super

child / sub

```cpp
int main()
{
    Male obj1;
    cout<<obj1.age<<' '<<obj1.weight<<' '<<obj1.height<<'\n';
    obj1.setweight(60);
    cout<<obj1.age<<' '<<obj1.weight<<' '<<obj1.height<<'\n';
    cout<<obj1.color<<'\n';
    obj1.sleep();

    return 0;
}
```

Received Output: Copy

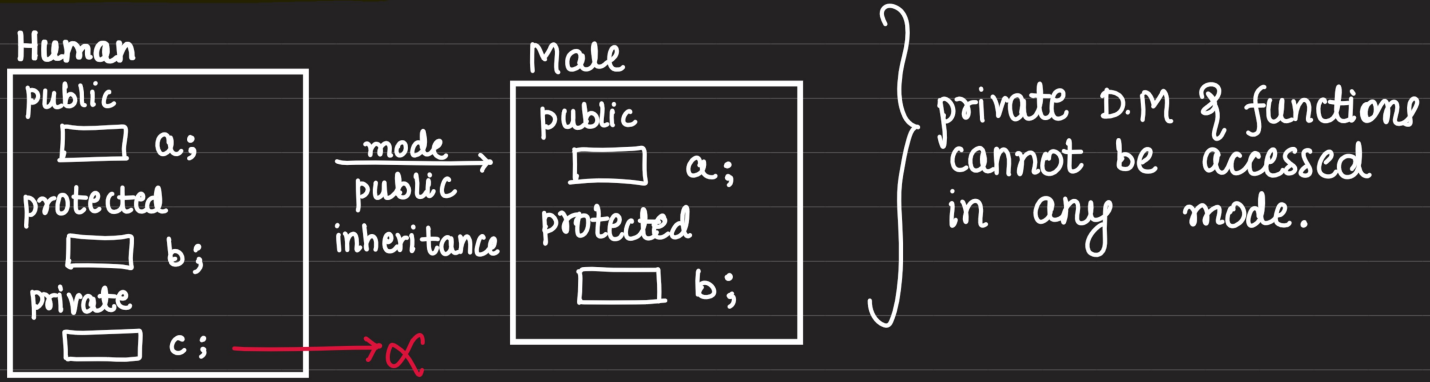
```
0 0 10
0 60 10
Black
Male is Sleeping
```

Mode of Inheritance

↳ public ↳ private ↳ protected

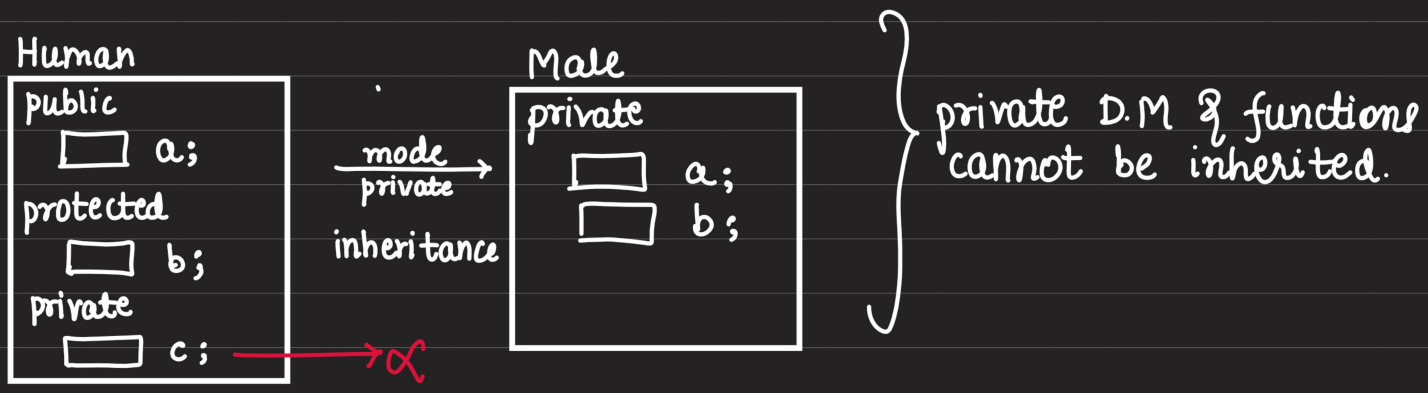
Base Class member Access Specifier (Parent class)	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible	Not Accessible	Not Accessible

Public Inheritance



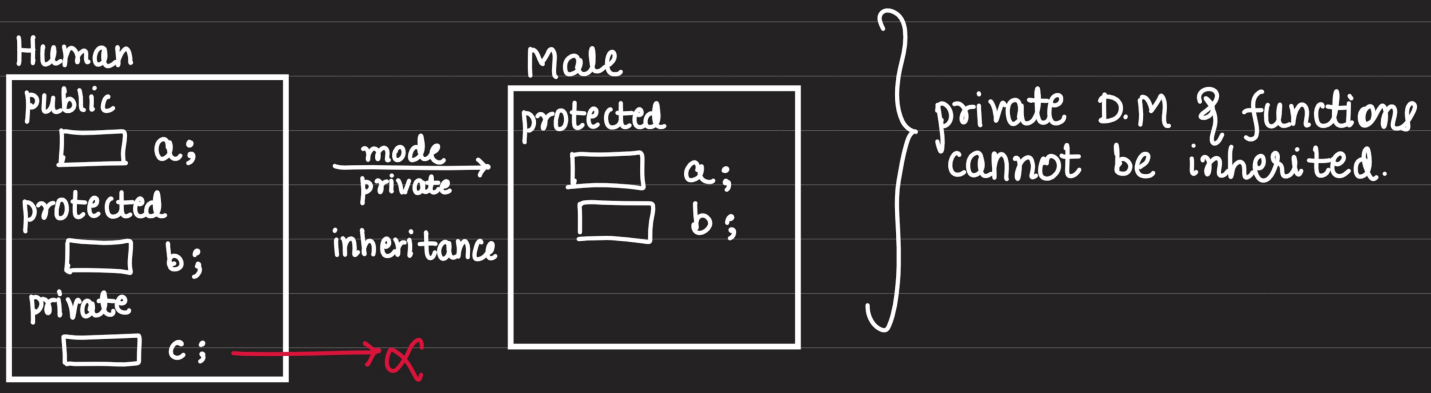
Only public / protected DM & functions are inherited and they remains public/protected in child class.

Private Inheritance



Only public / protected DM & functions are inherited and they are changed to private in child class.

Protected Inheritance



Only public / protected DM & functions are inherited and they are changed to protected in child class.

```
class Human{
public:
    int height;
    int weight;
    int age;
    Human(){
        height = weight = age = 0;
    }
    int getAge(){ return this->age; }
    int setweight(int w){ this->weight = w; }

};
class Male: protected Human {
public:
    string color;
    Male(){
        height = 10;
        color = "Black";
    }
    int getHeight(){return this->height;}


};
```

```
int main()
{
    Male obj1;
    cout<<obj1.age<<' '<<obj1.weight<<' '<<obj1.height<<'\n';
    obj1.setweight(60);
    cout<<obj1.color<<'\n';
    obj1.sleep();

    return 0;
}
```

cannot access protected

```
int main()
{
    Male obj1;
    cout<<obj1.getHeight()<<'\n';

    return 0;
}
```

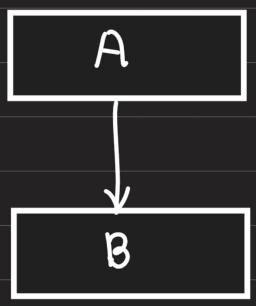
we will need to write a getter.

- protected is similar to private class but it can be inherited

Types of Inheritance

i) Single
ii) Multi-level
iii) Multiple
iv) Hierarchical
v) Hybrid

i) Single Inheritence



Inheritance_type

TC 1
Failed 88ms

Input: Copy

Expected Output: Copy

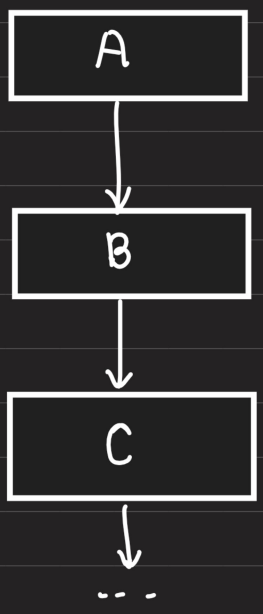
Received Output: Copy
Speaking

+ New Testcase

Run All + New

```
4   class Animal{
5   public:
6       int weight;
7       int age;
8       void speak(){
9           cout<<"Speaking\n";
10      }
11
12  };
13  class Dog: public Animal {
14
15  };
16
17  int main()
18  {
19      Dog d;
20      d.speak();
21      return 0;
22  }
```

ii) Multilevel :



Local:
Inheritance_types

TC 1
Failed 31ms

Input: Copy

Expected Output: Copy

Received Output: Copy
Speaking
Bark!

+ New Testcase

Set ONLINE_JUDGE

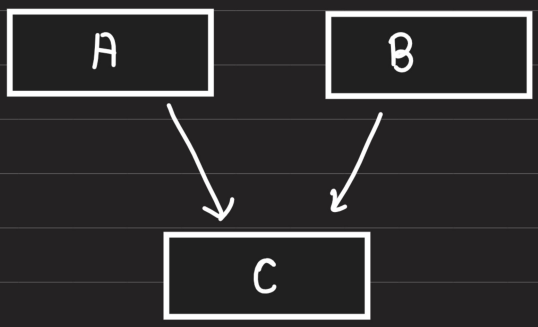
Support

Feedback

Run All + New

```
4   class Animal{
5   public:
6       int weight;
7       int age;
8       void speak(){
9           cout<<"Speaking\n";
10      }
11
12  };
13
14  class Dog: public Animal {
15  public:
16      string sound = "Bark!";
17  };
18
19  class GermanShepher: public Dog{
20
21  };
22
23  int main()
24  {
25      GermanShepher d;
26      d.speak();
27      cout<<d.sound<<'\n';
28      return 0;
29  }
```

iii) Multiple Inheritance



C inherits multiple parent class.

Inheritance_types

TC 1 Failed 25ms

Input: Copy

Expected Output: Copy

Received Output: Copy
Speaking
HOMO-SAPIENS

+ New Testcase

Set ONLINE_JUDGE

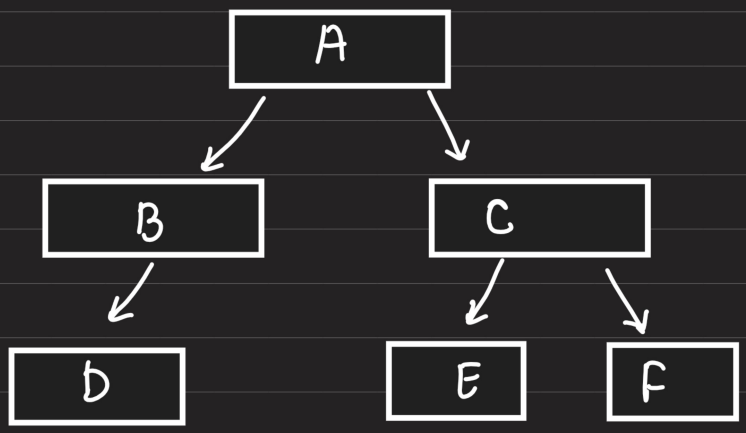
Support

Feedback

Run All | + New | Stop | Help | Delete

```cpp
class Animal{
public:
    int weight;
    int age;
    void speak(){
        cout<<"Speaking\n";
    }

};

class Human{
public:
    string color = "";
    void species(){
        cout<<"HOMO-SAPIENS\n";
    }
};

class Hybrid: public Animal, public Human{

};

int main()
{
    Hybrid obj;
    obj.speak();
    obj.species();
    return 0;
}
```

iv) Hierarchical



single parent is inherited by multiple child classes

Local: Hierarchical_Inhe

TC 1 Failed 23ms

Input: Copy

Expected Output: Copy

Received Output: Copy
Function 1
Function 2

Function 1
Function 3

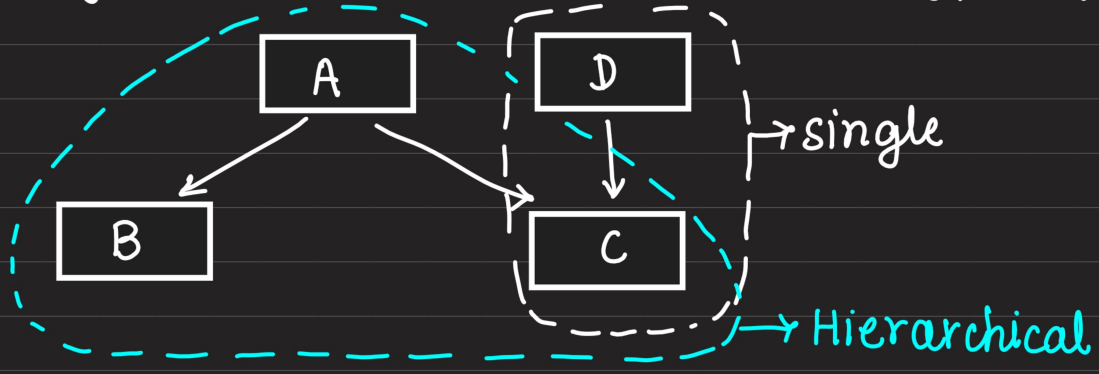
+ New Testcase

Set ONLINE_JUDGE

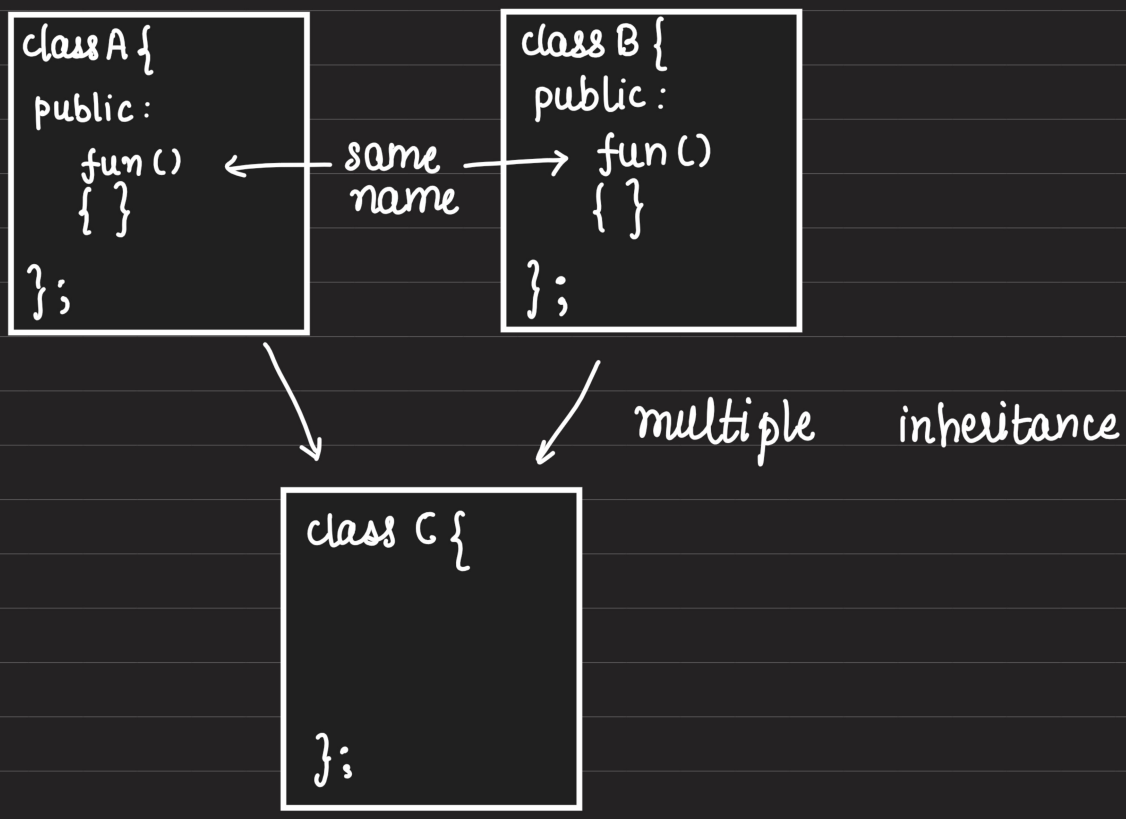
Run All | + New | Stop | Help | Delete

```cpp
class A{
public:
    void func1(){
        cout<<"Function 1\n";
    }
};

class B: public A{
public:
    void func2(){
        cout<<"Function 2\n";
    }
};
class C: public A{
public:
    void func3(){
        cout<<"Function 3\n";
    }
};

int main()
{
    B obj2;
    obj2.func1(); obj2.func2();
    cout<<'\n';
    C obj3;
    obj3.func1(); obj3.func3();
}
```

v) Hybrid : Combination of above types of inheritance



Inheritance Ambiguity



C obj; obj.fun() } → which fun is this (of class A or class B)

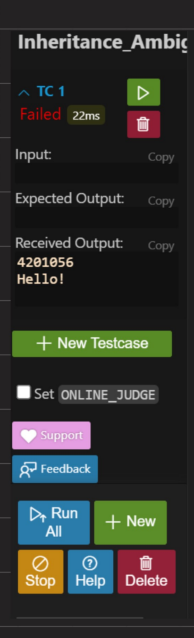
→ To remove this ambiguity, we use :: (scope resolution operator)

Call it as

obj.A::fun() / obj.B::fun()

```cpp
class Dog{
public:
    int weight, age;
    void speak(){
        cout<<"BARK\n";
    }

};

class Human{
public:
    int weight, age;
    void speak(){
        cout<<"Hello!\n";
    }
};

class mul_inheri: public Dog, public Human{

};

int main()
{
    mul_inheri obj;
    obj.age;
    obj.speak();
    return 0;
}
```

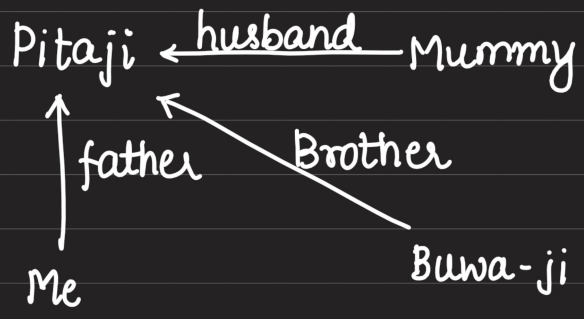
```
Inheritance_Ambiguity.cpp:28:9: error:
 request for member 'age' is ambiguous
   28 |     obj.age;
      |         ^~~
Inheritance_Ambiguity.cpp:15:17: note:
 candidates are: 'int Human::age'
   15 |     int weight, age;
      |                 ^~~
Inheritance_Ambiguity.cpp:6:17: note:
                 'int Dog::age'
    6 |     int weight, age;
      |                 ^~~
Inheritance_Ambiguity.cpp:29:9: error:
 request for member 'speak' is ambiguo
us
   29 |     obj.speak();
      |         ^~~~~
Inheritance_Ambiguity.cpp:16:10: note:
 candidates are: 'void Human::speak()'
   16 |     void speak(){
      |          ^~~~~
Inheritance_Ambiguity.cpp:7:10: note:
                 'void Dog::speak()'
    7 |     void speak(){
      |          ^~~~~
```




```cpp
class Dog{
public:
    int weight, age;
    void speak(){
        cout<<"BARK\n";
    }

};

class Human{
public:
    int weight, age;
    void speak(){
        cout<<"Hello!\n";
    }
};

class mul_inheri: public Dog, public Human{

};

int main()
{
    mul_inheri obj;
    cout<<obj.Dog::age<<'\n';
    obj.Human::speak();
    return 0;
}
```

Polymorphism : existing in multiple form.

many forms



Polymorphism Types

i) Compile time polymorphism / Static polymorphism
ii) Run time polymorphism / Dynamic polymorphism

i) Compile time polymorphism

- Function overloading
- operator overloading

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

class A{
public:
    void sayHello(){
        cout<<"Hello Shahil\n";
    }
    void sayHello(){
        cout<<"Hello Shahil\n";
    }
};

int main()
{
    A obj;
    obj.sayHello();
    return 0;
}
```

```
PS C:\Users\shahi\Desktop\CDC\OOPs> cd "c
:\Users\shahi\Desktop\CDC\OOPs\Pillers_of
_OOPs\" ; if ($?) { g++ Polymorphism.cpp
-o Polymorphism } ; if ($?) { .\Polymorph
ism }
Polymorphism.cpp:9:10: error: 'void A::sa
yHello()' cannot be overloaded with 'void
 A::sayHello()'
    9 |     void sayHello(){
      |          ^~~~~~~~
Polymorphism.cpp:6:10: note: previous dec
laration 'void A::sayHello()'
    6 |     void sayHello(){
      |          ^~~~~~~~
PS C:\Users\shahi\Desktop\CDC\OOPs\Piller
s_of_OOPs>
```

During compile-time the code gives error.

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

class A{
public:
    void sayHello(){
        cout<<"Hello Shahil\n";
    }
    void sayHello(string name){
        cout<<"Hello "<<name<<'\n';
    }
};

int main()
{
    A obj;
    obj.sayHello();
    obj.sayHello("Shahil");
    return 0;
}
```

```
PS C:\Users\shahi\Desktop\CDC\OOPs> cd "c:
\Users\shahi\Desktop\CDC\OOPs\Pillers_of_O
OPs\" ; if ($?) { g++ Polymorphism.cpp -o
Polymorphism } ; if ($?) { .\Polymorphism
}
Hello Shahil
Hello Shahil
PS C:\Users\shahi\Desktop\CDC\OOPs\Pillers
_of_OOPs>
```

If we change the signature of the function, it works.

↳ This way we can overload the function.

Changing function type, without changing the parameters cannot overload the function.

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

class A{
public:
    void sayHello(){
        cout<<"Hello Shahil\n";
    }
    int sayHello(){
        return 0;
    }
};

int main()
{
    A obj;
    obj.sayHello();
    return 0;
}
```

```cpp
class A{
public:
    void
    }
    int sayHello(){
        return 0;
    }
};
```

cannot overload functions distinguished by return type alone C/C++(311)

inline int A::sayHello()

View Problem (Alt+F8) Quick Fix... (Ctrl+.)

But changing input Parameter will make it overloading.

```cpp
class A{
public:
    void sayHello(){
        cout<<"Hello Shahil\n";
    }
    int sayHello(string name){
        cout<<"Hello "<<name<<'\n';
        return 0;
    }
};

int main()
{
    A obj;
    obj.sayHello();
    cout<<obj.sayHello("Shahil");
    return 0;
}
```

```
of_OOPs\" ; if ($?)
{ g++ Polymorphism.c
pp -o Polymorphism }
 ; if ($?) { .\Polym
orphism }
Hello Shahil
Hello Shahil
0
PS C:\Users\shahi\De
sktop\CDC\OOPs\Pille
rs_of_OOPs>
```

#Concept : For function overloading, i.e, if we want to create multiple functions using the same name, we will have to change the input parameters.

No two of them can have the same input argument.

Either have different no. of argument / different type of argument.

Operator overloading

+ ⟶ add or concatenate

→ print ("Shahil")

→ instead of adding, do substraction

} overloading

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

} these operators can be overloaded

Operators that cannot be overloaded are :
i) :: ii) . iii) .* iv) :?

Syntax for overloading '+'

return_type operator+ ( ) ← i/p argument
{
}

a + b ← i/p argume.
↑ current object

Local:
Operator_Overload

TC 1
Failed 20ms

Input: Copy

Expected Output: Copy

Received Output: Copy
output 3
Main Parenthesis hoon4

+ New Testcase

Set ONLINE_JUDGE

Support Feedback

```
4   class A{
5   public:
6       int a,b;
7       int add(){
8           return a+b;
9       }
10      void operator+ (A &obj){
11          int val1 = this->a;
12          int val2 = obj.a;
13          cout<<"output "<<val2-val1<<'\n';
14      }
15      void operator() (){
16          cout<<"Main Parenthesis hoon"<<this->a<<"\n";
17      }
18  };
19
20  int main()
21  {
22      A obj1, obj2;
23      obj1.a = 4;
24      obj2.a = 7;
25      obj1+obj2;
26      obj1();
27  }
```

ii) Run time polymorphism

Whenever an object is bound with the functionality at run time.

Method overriding

```
class Animal
{
  public:
    void speak()
    {
      cout << "Speaking";
    }
};
```

inherited →

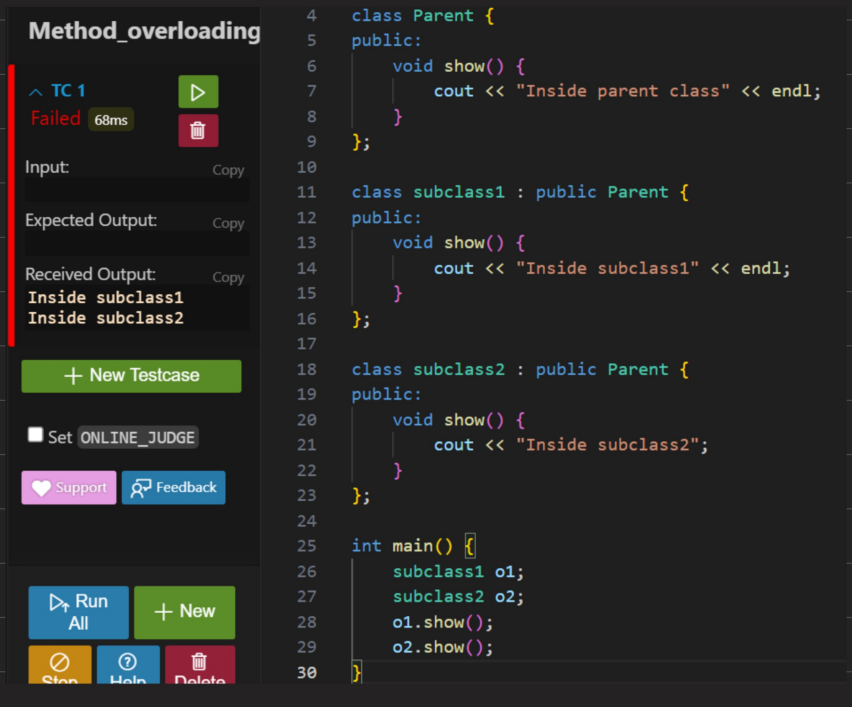
```
class Dog : public Animal
{
  public:
    void speak()
    {
      cout << "Barking";
    }
};
```

name of function is same as function inherited from pattern

This is method overriding

Rules for Method overriding

- The parent class method and the method of the child class must have the same name.
- The parent class method and the method of the child class must have the same parameters.
- It is possible through inheritance only.



this overrides the parent function.

Abstraction

↳ Implementation hiding. We show only essential stuff.

Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

Real-life example: When you send an email to someone, you just click send, and you get the success message; what happens when you click send, how data is transmitted over the network to the recipient is hidden from you (because it is irrelevant to you).

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data members will be visible to the outside world and not. Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members.

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

class abstraction {
private:
    int a, b;

public:
    // method to set values of private members
    void set(int x, int y) {
        a = x;
        b = y;
    }

    void display() {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main() {
    abstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```