OOPS c++

Heap
stack ← limited

Green City®
Date: __.__.____

→ In c++ objects can be created by two ways :-

i) Statically
ii) Dynamically

*Statically

→ when we declare an object normally (without new), it is created on the stack (automatic storage)

→ These objects are automatically destroyed when they go out of space

* Dynamically

→ when we allocate an object using new, it is created in the heap

→ These object persist until you explicitly delete them using delete.

ex- student *s = new student;

Access

(*s).name = "Rohit";

s→name = "Rohit";

ex-
```
#include<iostream>
using namespace std;
class student
{
public:
```

```
string name;
int age, roll_number;
string grade;
};

int main()
{
  student *s = new student;
  (*s).name = "Rohit";
  (*s).age = 10;
    :
    :
  cout << s->name << "\n";
```

Q Why there is need of dynamic allocation of object?

→ :-
- when object size is too large for stack.
- when object lifetime must extend beyond scope
- when the number of objects is unknown at compile time.

Date: __.__.____

Constructor:

→ It is a special function that is invoked automatically at the time of object creation

→ Name of the constructor should be same as class name.

→ It doesn't have any return type.

→ It is used to initialize the value

EX—

```
class customer {
String name;
int ac-no;
int balance;
customer ()          // Default constructor
{
cout<<" customer is called";
}
customer (string a, int b, int c) // Parameterized constructor.
{
name = a ;
ac-no = b ;
balance = c ;
}
};
int main ()
{ customer A1 ("Kaushal", 1234, 1000);
```

Date:

this stores the address of object from where they called.

// Parameterised constructor.

```
customer (string name, int ac-no, int balance)
{
this → name = name;
this → ac-no = ac-no;
this → balance = balance;
```

+ if the name of constructor is same but they will take different parameters or arguement then this is called constructor overloading. Here class is same.

// inline constructor.

```
inline customer (string a, int b, int c):
name(a), account-no(b), balance(c) {
}
```

+ Both constructor and Destructor are assigned in public.

Date: __.__.____

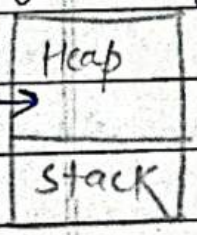
Destructor :

→ It is an instance member function that is invoked automatically whenever an object is going to be destroyed.

→ It is the last function that is going to be called before an object is destroyed.

```
class customer {
    string name;
    int * balance;

    customer (string name, int balance)
    {
        this → name = name;
        balance = new int;
        * blance = balance;
    }
~ customer ()
    { delete balance;
      cout << "destructor is called";
    }
};

int main()
{
    customer A1("Rohit", 1000);

}
```

Here memory is dynamically assigned.



Date: __.__.____

- By default destructor is called after the program execution, it will destroy the memory space of stack section.
- If we declare destructor by own then it is called after program execution and it will release the memory space that is dynamically assigned.
- constructor is called orders wise (asce)

```
int main()
{
    customer A1("1"), A2("2"), A3("3");
}
```

output:

1
2
3

- Destructor is called Reverse order (Desc.)

output

3
2
1

Green City

Date: __.__.____

→ Shallow copy Vs Deep copy.

1. Shallow copy:
   - creates a new object, but does not create copies of nested objects (Reference are shared).
   - changes made to nested objects in the copied version also reflect in the original.

Ex- .py

```
original_list = [[1,2,3],[4,5,6]]
shallow_copied_list = copy.copy(original_list)
shallow_copied_list[0][0] = 99
original_list = [[99,2,3],[4,5,6]]
```

2. Deep copy
   - creates a new object and recursively copies all nested objects (no shared references).
   - changes made in the deep-copied version do not affect the orginal.

ex-

```
deep_copied_list = copy.deepcopy(original_list)
deep_copied_list[0][0] = 42

print(original_list)  [[99,2,3],[4,5,6]]

print(deep_copied_list) [[42,2,3],[4,5,6]]
```

Date:

shallow copy	Deep copy
– creates a new object.	– creates the new object
– Reference are shared.	– completly copies (new reference)
– More efficient (less memory)	– uses more memory (full duplication)
– Affects the original	– Does not affect the original.

STATIC DATA MEMBER.

– They are attribute of classes or class member. It is declared using static Keyword.

– Only one copy of that member is created for the entire class & is shared by all objects.

– It is initialized before an objects of this class is created.

ex- class ____ {

____

static int ____ ;

};

int class ____ :: ____ = 0;

Green City
Date: __.__.____

- We can access by classname :: static member
  # static data member must be in public

- If the static data member is in Polvate then we have to make static member function to access

Ex- Public:

static void accessfare()

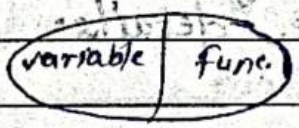
{

cout ____

}

ENCAPSULATION.

- Wrapping up data & information in a single unit, while controlling Access to them.



→ It means that we cannot direct Access that variable.

class

- To make any changes in variable we must have to create function to access or change.

Here comes the concept of data hiding.

Data hiding means that → By mistake user give wrong input, here by using data hiding feature wrong input can't change in original code.

Ex.- age -10, etc.

Date: __/__/____

Advantages:—

→ Data security — Prevent Unauthorized Access.

→ Code Maintainability — changes in implementation don't affect external code

→ Data validation — Ensure only valid data is stored.

→ Better Modularity — Helps in organizing code efficiently.

ABSTRACTION

→ Displaying only essential information & hiding the details.

Advantages:-

→ Hides complexity : Users interact with high-level classes without worrying about implementation details.

→ Increases flexibility : code can be modified without affecting user.

→ Essential for large Projects: commonly used in real-world application like Banking system, ATM etc.
→ inner mechanism are hidden.

INHERITANCE

→ Inheritance is an object-oriented programming (OOP) principle that allows one class (child/derived class) to acquire the properties and behaviours of another class (Parent class/ base class). It promotes code reusability, hierarchy, and extensibility.

ADVANTAGES:

→ Code Reusability – Avoids redundant code.

→ Better organization – Establishes a hierarchy in OOP.

→ Extensibility – Allows adding new features.

→ Support Polymorphism – Derived class can override Parent class methods.

* Types of Inheritance.

1. Single Inheritance – One child inherit from one Parent.

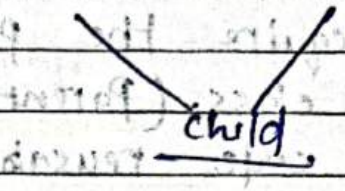
Ex – Animal (Parent/base class)
↓
Dog (Derived / child class)

Problems:- Limited code Reusability.

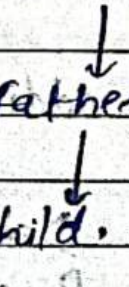
Ex – A Bird class have flight abilities, An animal class has general behaviours but a 'Bat' needs both, it can't inherit from both in single inheritance.

Date: __.__.____

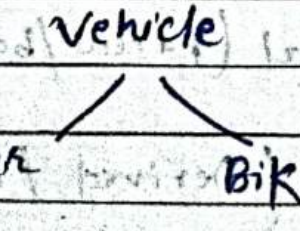
2. Multiple Inheritance – A child class inherits from multiple parent classes.

Ex – 

3. Multilevel Inheritance – A child class inherits from another child class.

Ex – 

4. Hierarchical Inheritance – Multiple child classes inherit from a single parent.

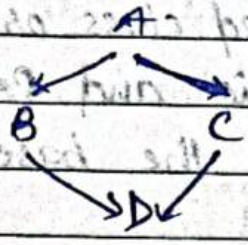
Ex – 

Green City

Date: __.__.____

5. Hybrid inheritance - A combination of two or more types of inheritance.

ex -



Access specifiers.

→ It is also known as access modifiers, they are the keywords used to define the accessibility of classes, methods, and variables. They control how the members of a class can be accessed from other classes.

	can use by external code	can use within a class	can use by derived class
Public	✓	✓	✓
Protected	X	✓	✓
Private	X	✓	X

— constructor called in increasing order
first to last.

— Destructor called in decreasing order.
last to first

Date:__.__.____

Single Inheritance

- In c++, single inheritance refers to - where a derived class or child class inherits the public and protected (functions and variables) of the base class or parent class.
- This helps the reuse of code of the base class.

Key points:

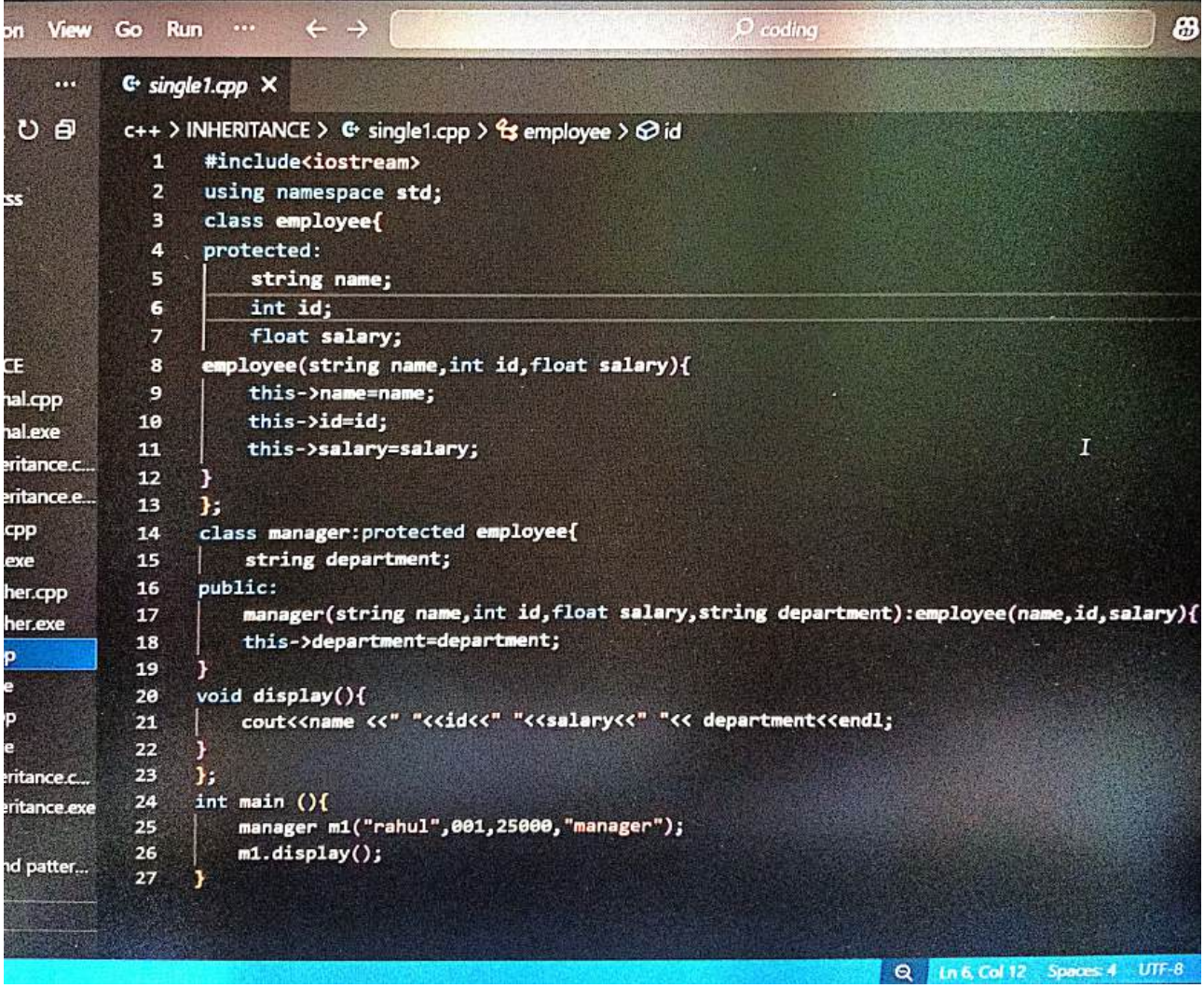
- Derived class can only access to the public and protected member of base class.
- Private member of base class is not directly accessible.
- Derived class can add its own member (variables, functions).
- You can also override base class methods in the derived class.

Go Run ··· ← → coding

singleinheritance.cpp ×

c++ > INHERITANCE > singleinheritance.cpp > person > person(string, int)

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

class person{
 protected:
        string name;
        int age;
    person(string name , int age){
        this->name=name;
        this->age=age;
    }
};
class student :public person{
        string grade; // this is by deafault private
 public:
    student(string name , int age ,string grade):person(name,age){
     this->grade=grade;
    }
    void display(){
        cout<<name<<" "<<age<<" "<<grade<<" ";
    }
};
int main (){
    student s1("kaushal",10,"A");
    s1.display();
}
```

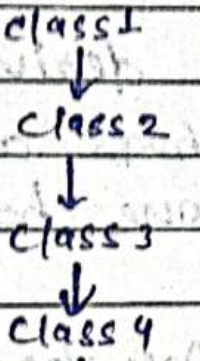


Multilevel Inheritance.

In c++ multilevel inheritance refers to where a class is derived from another class, which is also derived from another class.

concept:

1. Base class
2. Derived class
3. further derived class.



Syntax.

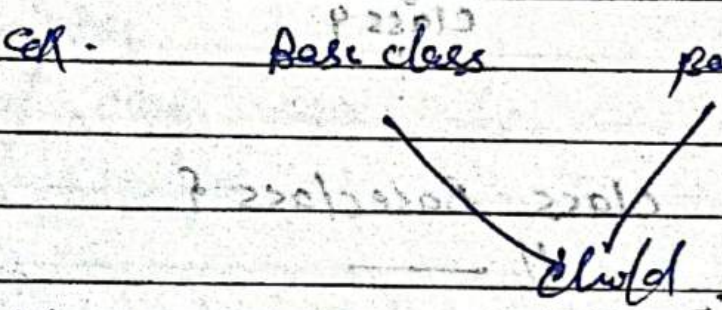
```
class Baseclass {
//
};

class Derivedclass1 : public Baseclass {
//
};

class Derivedclass2 : public Derivedclass1 {
//
};
```

Date:__.__.____

Multiple Inheritance

- In c++ Multiple Inheritance refers to the ability of a class to inherit from more than one base class.

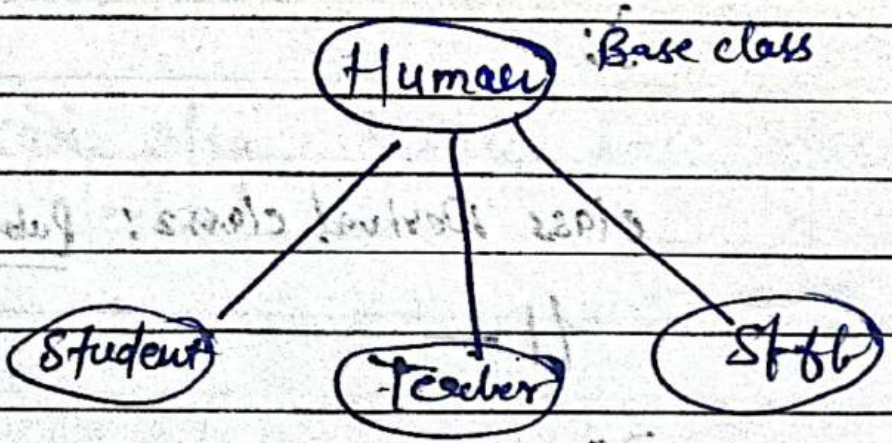
This allows a derived class to access members (methods and variables) of more than one base class.



class child : Pub Base1 , Pub Base2 {

};

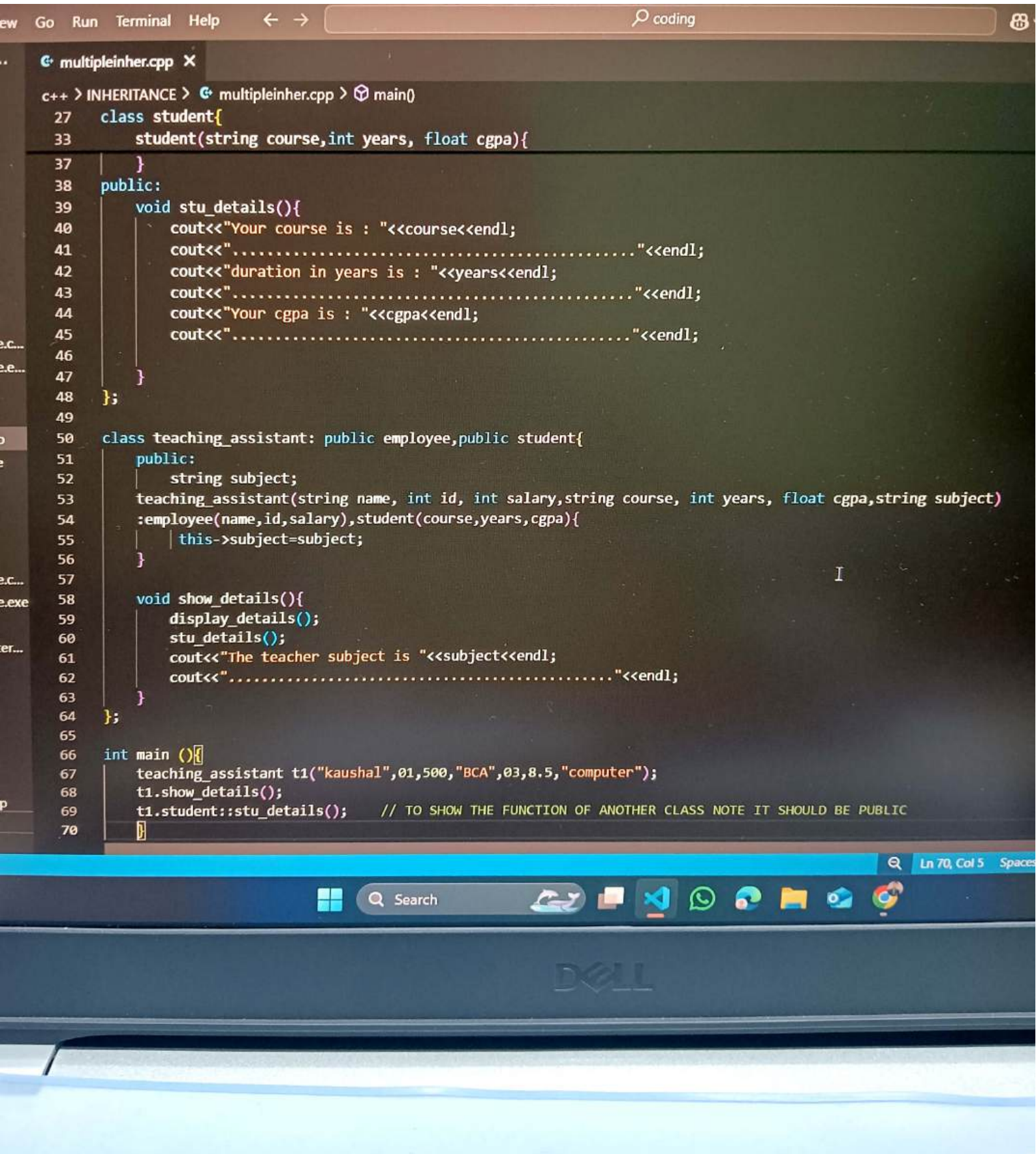
Hierarchical Inheritance



multipleinher.cpp

c++ > INHERITANCE > multipleinher.cpp > student > stu_details()

```cpp
#include<iostream>
using namespace std;
class employee{                    //MULTIPLE INHERITANCE
protected:
    string name;
    int id;
    int salary;

    employee(string name,int id, int salary){
        this->name=name;
        this->id=id;
        this->salary=salary;
    }

    void display_details()
    {
        cout<<"......................................................"<<endl;
        cout<<"Your name is : "<<name<<endl;
        cout<<"......................................................"<<endl;
        cout<<"Your id is : "<<id<<endl;
        cout<<"......................................................"<<endl;
        cout<<"Your salary is : "<<salary<<endl;
        cout<<"......................................................"<<endl;
    }
};

class student{
protected:
    string course;
    int years;
    float cgpa;

    student(string course,int years, float cgpa){
        this->course=course;
        this->years=years;
        this->cgpa=cgpa;
    }
```

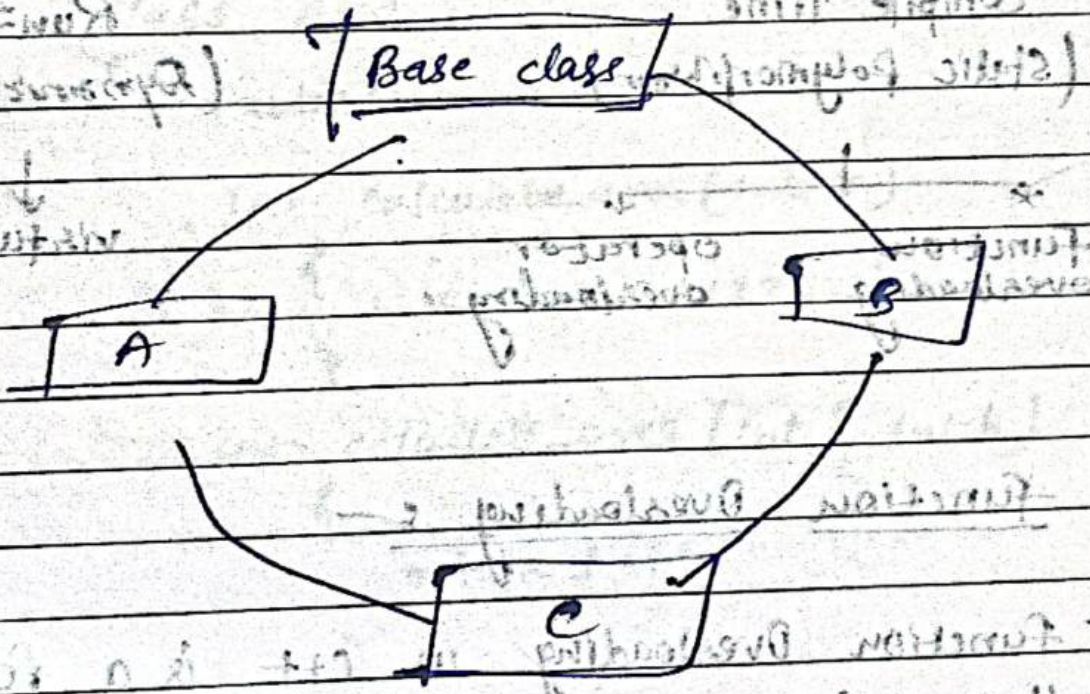


Green City

Date: __.__.____

Hybrid Inheritance. Multipath Inherit

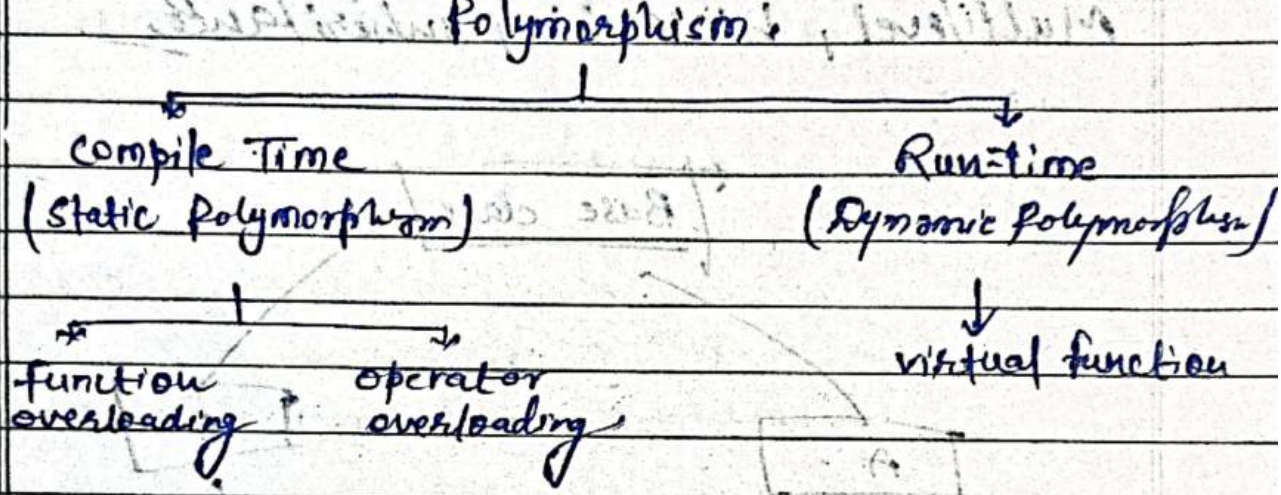
- It is a complex form of inheritance in object oriented programming (OOP). In hybrid inheritance, multiple types of inheritance are combined within a single class.
- In hybrid inheritance, within the same class we can have element of single, Multiple, Multilevel, hierarchical inheritance.



Polymorphism :-

→ Polymorphism is one of the core concepts of object-oriented programming (OOP) in C++.

→ It allows objects of different types to be treated as object of common base type.



function Overloading :-

→ function Overloading in C++ is a feature that allows you to define multiple functions with the same name but with different parameters (either in number or type).

→ It is a form of compile-time polymorphism, where the function to be called is determined at compile time based on the number or types of arguments passed to it.

Date: __/__/____

Key points about function overloading:-

1. Same function name.
2. Different parameters.
3. Return type doesn't matter.

Syntax for function overloading -

return_type function_name (parameter_list);

Ex:-

```
class Area
{ public:
    int calculate_area (int r)
    {
        return 3.14 * r * r;
    }
    int calculate_area (int l, int b)
    {
        return l * b;
    }
};
int main ()
{
    Area A1, A2;
    A1. calculate_area (4);
    A2. calculate_area (3,4);
}
```

Gran City

Function Overloading

Date: __.__.____

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

class Print {
public:
    // overloaded function to print an integer
    void display (int i) {
        cout << "Integer: " << i << endl;
    }
    // overloaded function to print a double
    void display (double d) {
        cout << "Double: " << d << endl;
    }
    // overloaded function to print a string
    void display (string s) {
        cout << "String: " << s << endl;
    }
};

int main () {
    Print obj;

    obj.display (5);    // calls display (int)

    obj.display (3.14);    // calls display (double)

    obj.display ("Hello");    // calls display (string).

    return 0;
}
```

Green City
Date: __.__.____

Operator Overloading

→ operator can also be a function name:

```
return type operator symbol (Parameter list)
{
  //_____
}
```

→ It is a feature that allows us to redefine the functionality of operator (+, -, *) etc for user-defined types (like classes).

→ In c++, you can overload an operator by providing a function that specifies how the operator should behave when applied to object of a class.

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

class complex
{
    int real, img;
public:
    complex ()
    {
    };

    complex (int real, int img)
    {
        this -> real = real;
        this -> img = img;
    };
```

```
void display ()
{
   cout <<real <<" +i " <<img <<endl;
}
complex operator + (complex &c)
{
    complex ans;
    ans.real = real + c.real;
    ans.img = img + c.img;
    return ans;
}
};
int main()
{
   complex c1(3,4);
   complex c2(4,6);

   complex c3 = c1+c2;
   c3.display();

};
```

→ You cannot overload → ::, ., .*, ?:, sizeof, typeid, alignof, noexcept.

Date: __.__.____

Virtual function

→ A virtual function in c++ is a function that is defined in a base class.

→ When a function is declared as virtual, it tells the c++ compiler to support dynamic polymorphism (runtime polymorphism).

→ Dynamic Dispatch: The decision about which function to call is made at runtime, not at compile time. This is also known as late binding

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

class Animal {
public:
    virtual void sound ()
    {
        cout<<" Animal makes sound" <<endl;
    }
};

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

```
class cat : public Animal {
  public:
      void sound () {
        cout << "cat meows" <<endl;
      }
};

int main () {
  Animal *animalptr;

  animalptr = new Dog();
  animalptr -> sound();

  animalptr = new cat();
  animalptr->sound();

  return 0;
}
```

→ Overriding : The function in the derived class should match the signature of the base class function (same name, same parameter)

- function prototype includes:
  - Return Type
  - function name
  - Parameter Type and parameter name.

* function signature includes

- function name
- parameter Type. (but not parameter name).

Pure virtual function.

→ A pure virtual function is a function that is declared in base class and does not have any implementation in that class.

→ Pure virtual function is used to create Abstract class.

→ We do not create any object of Abstract class directly.

Syntax —

```
virtual return-type fun-name(parameters) = 0;
```

Green City
Date:__.__.____

Exception Handling

→ An Exception is an unexpected problem that arises during the execution of a program and our program terminates suddenly with some error/issues.

• Try • Catch • Throw.

→ It allows the program to detect and respond to runtime issues/errors without crashing code/program.

→ It uses Try, catch, and Throw to manage exceptions.

* The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.

* The catch statement represents a block of code that is executed when a particular exception is thrown from the try block.
The code to handle the exception is written inside the catch blocks.

Green City®
Date: __.__.____

→ An exception in c++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the throw exception.