Day 10 (11-12-2025)

Inheritance concept

- One class can inherit properties of another class
- Code reuse, runtime polymorphism, IS-A relationship
- Parent class/Base class/Super class
- Child class/Derived class/Sub class
- Syntax:

```
class Base{
    //members
};
class Derived : access_specifier Base{
    //members
};
```

Modes & types

Modes

- Public, Private and Protected
- Public -> all access via public/ inherit publically
- Protected -> all access via protected/ inherit protected
- Private -> all access private/inherit private
- Rules
    - Private members of base class are not accessible directly in derived class
    - Public members of base class become
        - public in derived class when inherited publicly
        - private in derived class when inherited privately
        - protected in derived class when inherited protectedly
    - Protected members of base class become
        - protected in derived class when inherited publicly
        - private in derived class when inherited privately
        - protected in derived class when inherited protectedly

Inheritance modes	Private members of Base class	Protected members of Base class	Public members of Base class
Public	Not accessible	Protected	Public
Private	Not accessible	Not accessible	Private
Protected	Not accessible	Protected	Protected

```cpp
class Base{
    int x = 10;
protected:
    int y = 20;
public:
    int z = 30;
};

class Derived : protected Base {    // by default private
public:
    void getY(){
        cout << y << endl;
    }

    void getZ(){
        cout << z << endl;
    }
};

class Derived1 : protected Derived{
public:
    void getY1(){
        cout << y << endl;
    }
};

int main() {
    Derived d1;

    //private mode inheritance
//  cout << d1.x << endl;   //not valid
//  cout << d1.y << endl;   //not valid
//  cout << d1.z << endl;   //not valid

    //public mode inheritance
//  cout << d1.x << endl;   //not valid
//  cout << d1.y << endl;   //not valid
    d1.getY();              //valid -> protected rule
//  cout << d1.z << endl;   //valid

    //protected mode inheritance
//  cout << d1.x << endl;   //not valid
//  cout << d1.y << endl;   //not valid
//  cout << d1.z << endl;   //not valid
    d1.getY();
    d1.getZ();

    Derived1 d2;

    d2.getY1();

    return 0;
}
```

Types

Single Inheritance

- One class inherit another class -> Parent -> Child

```
//class  A; //base class/parent class/super class
class A {
public:
    int x;
};

//class  B : A; //derived class/child class/sub class
class B : public A {
public:
    int y;

};
```

Multilevel

- One class inherit another class and again will be inherited by some other class -> GP -> P -> C

```
class A {
public:
    int x;
};
class B : public A {
public:
    int y;

};
class C : public B {
public:
    int z;
};
```

Hierarchical

- One class have multiple child class -> P -> C1, P -> C2

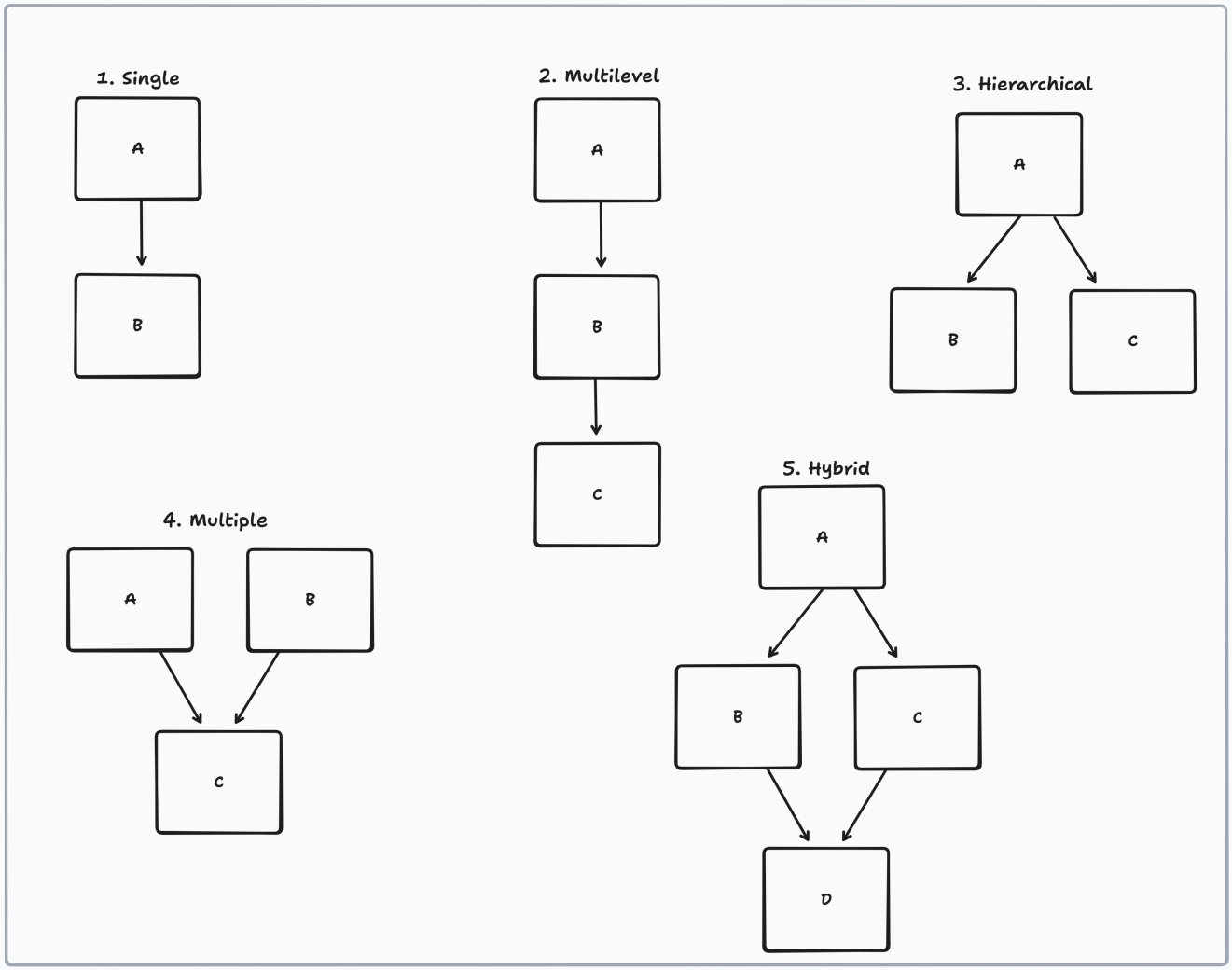
```
class A {
public:
    int x;
};
```

```
class B : public A {
public:
    int y;
};
class C : public A {
public:
    int z;
};
```

Multiple

- One class inherits from 2 different classes -> A, B -> C
- Hybrid -> B <- A, C <- A : D <- B, C

```
class A {
public:
    int x;
};
class B : public A {
public:
    int y;

};
class C : public A {
public:
    int z;
};
class D : public B, public C {
public:
    int a;
};
```



Diamond problem

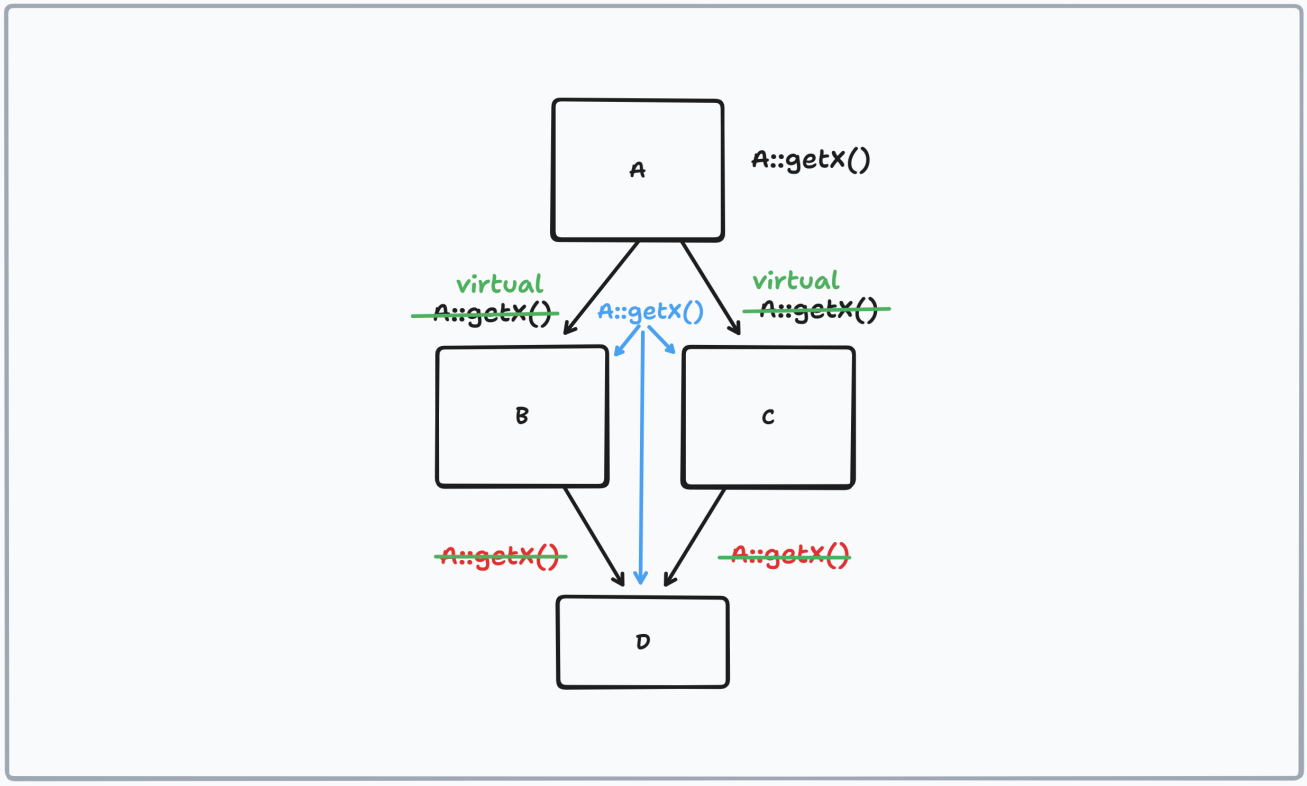
- Problem in multiple inheritance -> ambiguity
- When both base classes have same member and derived class try to access that member -> ambiguity
- Occurs when 2 classes inherited in a single class -> hybrid
- Solution -> `virtual base class`

Virtual base class

- To avoid ambiguity in diamond problem -> virtual base class
- When we declare base class as virtual -> only one copy of base class will be inherited by derived class

```
class A {
public:
    int x = 10;

    void getX(){
        cout << "value from A: "<< x << endl;
    }
};

class B : virtual public A{
```

```
public:
    int y;

};

class C : virtual public A{
public:
    int z;
};

class D : public B, public C{
public:
    int a;
};

int main() {
    D d
//  cout << d.x << endl;        //error: request for member 'x' is ambiguous
//  cout << d.x << endl;
    d.getX();   // valid -> only one copy of base class A

    return 0;
}
```



Upcasting

- To convert derived class object to base class object
- Implicit upcasting -> done by compiler automatically
- Explicit upcasting -> done by programmer using cast operator

- Parent obj, Child obj : child obj -> Parent obj
- Example:

```
#include <iostream>
using namespace std;
class Person{
public:

    string name;
    int age;
    void work(){
        cout << "Person working..." << endl;
    }
};
class Student : public Person{
public:
    string courseName;

    void work(){
        cout << "Doing study..." <<endl;
    }
};

int main(){
    //value
    Student s;
    Person p = s;   //implicit upcasting

    //reference
    Student s1;
    Person &p1 = s1;    //implicitly upcasting(person)

    //address
    Student *s2 = new Student();
    Person *p2 = s2;    //implicitly upcasting(person)

    s1.work();  // s1::work()
    p1.work();  // p1::work()

    s2->work(); // s1::work()
    p2->work(); // p1::work()

    return 0;
}
```

Downcasting

- To convert base class object to derived class object
- Implicit downcasting -> not allowed
- Explicit downcasting -> done by programmer using cast operator
- Parent obj, Child obj : parent obj -> child obj

- Example:

```
#include <iostream>
using namespace std;
class Person{
public:

    string name;
    int age;
    void work(){
        cout << "Person working..." << endl;
    }
};
class Student : public Person{
public:
    string courseName;

    void work(){
        cout << "Doing study..." <<endl;
    }
};
int main(){
    Person p;
    //Student s = p;    //implicit downcasting -> not allowed

    //explicit downcasting
    Student s = (Student)p; // unsafe -> may cause runtime error

    Person p1;
    Student &s1 = (Student&)p1; // unsafe -> may cause runtime error

    Person *p2 = new Person();
    Student *s2 = (Student*)p2; // unsafe -> may cause runtime error

    s.work();   //s::work()

    s1.work();  //s1::work()
    s2->work(); //s2::work()

    return 0;
}
```

Object slicing

- when we do upcasting, there might be case when we pass the object as value -> object slicing will cause.
- derived class object -> upcasting in base class object -> it can loose some data or its behaviour changes.

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

class Person{
public:

    string name;
    int age;
    void work(){
        cout << "Person working..." << endl;
    }
};

class Student : public Person{

public:
    string courseName;

    void work(){
        cout << "Doing study..." <<endl;
    }
};

int main(){
    //value
    Student s;
    Person p = s;

    //reference
    Student s1;
    Person &p1 = s1;    //implicitly upcasting(person)
    //no runtime polymorphism

    //address
    Student *s2 = new Student();
    Person *p2 = s2;

    s1.work();  // s1::work()
    p1.work();  // p1::work()

    s2->work(); // s1::work()
    p2->work(); // p1::work()

    return 0;
}
```

Virtual function

- A function in base class which is declared with keyword `virtual`
- It is used for achieving runtime polymorphism
- To denote the behaviour will be decided at runtime for derived class object

Function overriding

- To provide specific implementation of base class function in derived class
- When a derived class has a function with same name and signature as in base class
- to change the implementation according to the use of class

Early/Late binding

Early binding

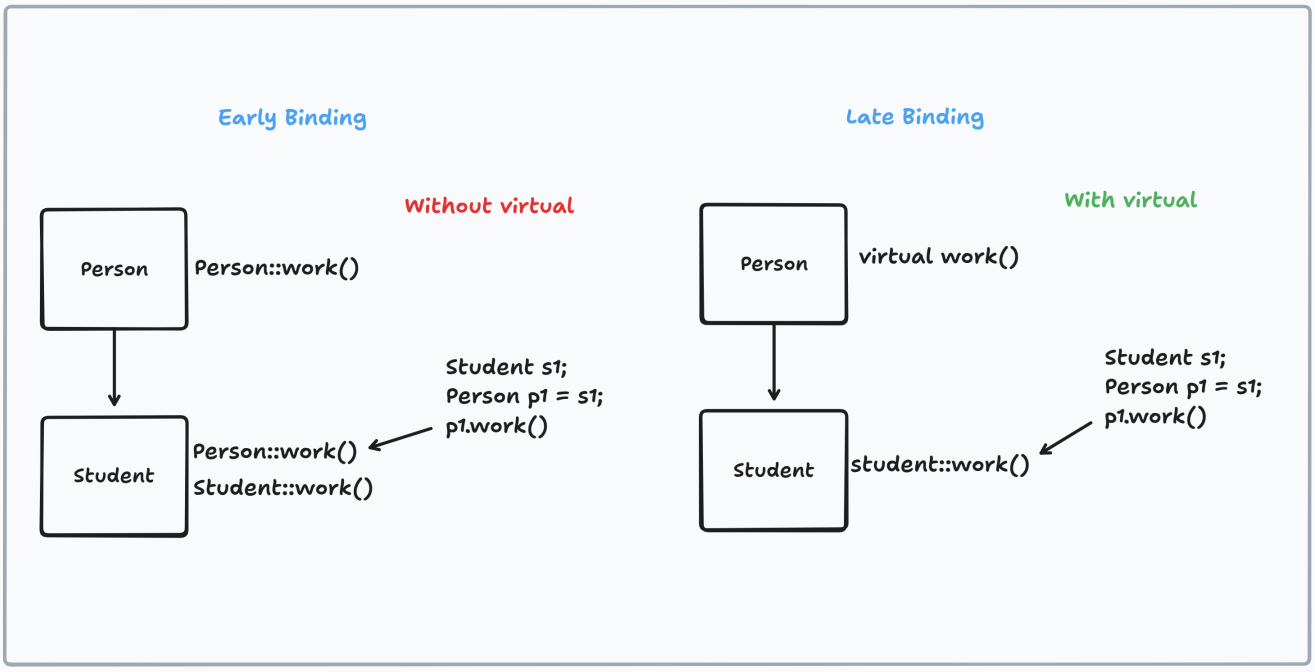
- compile time binding of object/methods
- Compile time polymorphism
- Function overloading

```
class Person{
public:

    string name;
    int age;
//  Person *this
    void work(){        // compile time polymorphism -> early binding
        cout << "Person working..." << endl;
    }
};

class Student : public Person{
public:
    string courseName;

//  Student *this;
    void work()  {
        cout << "Doing study..." <<endl;
    }
};

int main(){

    Person *p = new Student();  // upcast

    p->work();      // student

    return 0;
}
```

Late binding

- Runtime binding of object/methods
- Runtime Polymorphism
- Virtual function

```cpp
class Person{
public:

    string name;
    int age;
//  Person *this -> it will not be here
    virtual void work(){        // runtime polymorphism -> late binding
        cout << "Person working..." << endl;
    }
};
class Student : public Person{
public:
    string courseName;
//  Student *this;
    void work()  {
        cout << "Doing study..." <<endl;
    }
};
int main(){

    Person *p = new Student();  // upcast

    p->work();      // student

    return 0;
}
```

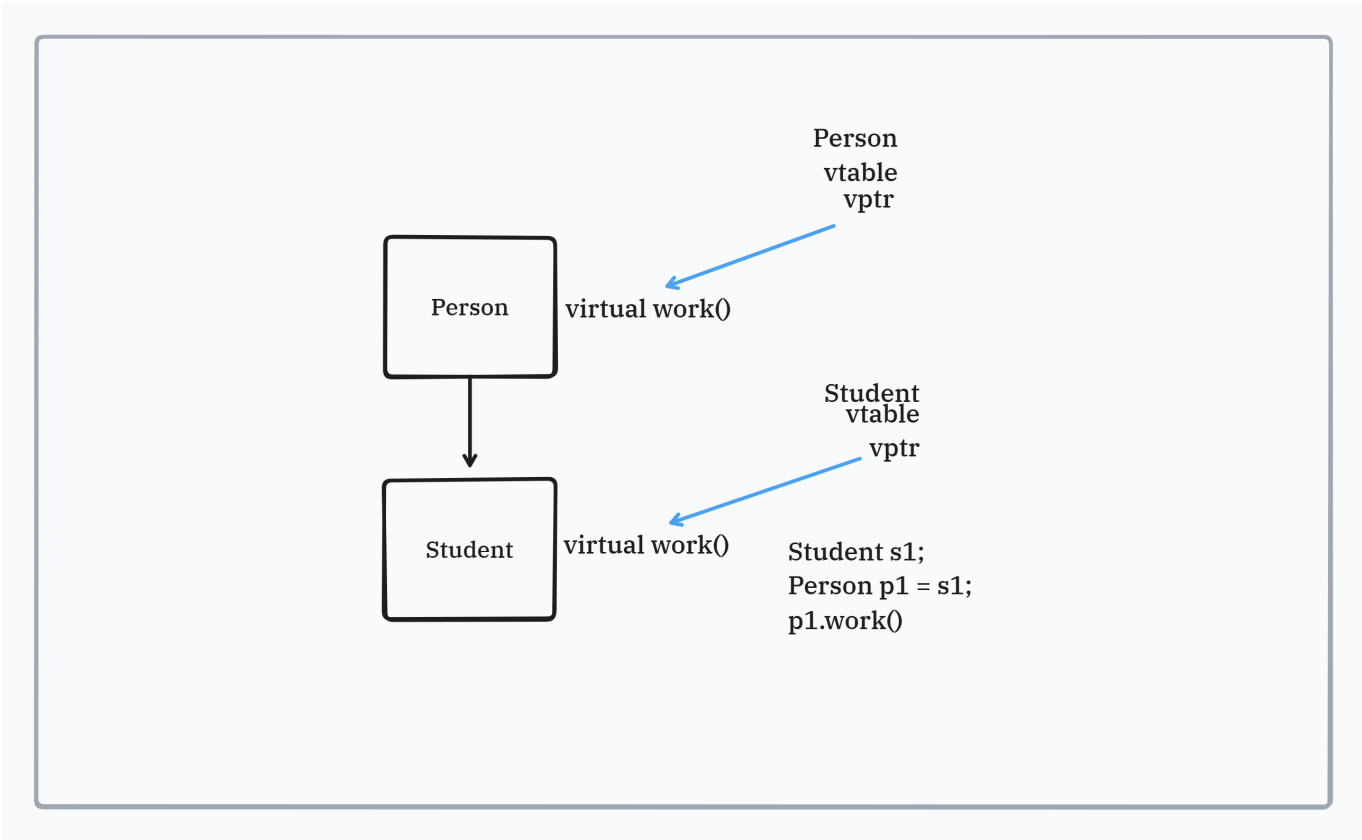


vtable/vptr

- Virtual function mechanism
- When a class has virtual function, compiler creates a table called vtable for that class

- vtable contains addresses of virtual functions of that class
- Each object of that class contains a pointer called vptr which points to the vtable of that class
- vtable -> virtual table -> virtual function
- vptr -> virtual pointer
- vptr -> work()
- vptr -> Person::work()

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

class Person{
public:

    string name;
    int age;
//  Person *this -> it will not be here
    virtual void work(){        // runtime polymorphism -> late binding
        cout << "Person working..." << endl;
    }

//  vtable -> virtual table -> one virtual function
//  vptr -> virtual pointer
    // vptr -> work()
    // vptr -> Person::work()
};

class Student : public Person{
public:
    string courseName;

//  Student *this;
    void work()  {
        cout << "Doing study..." <<endl;
    }
//  vtable -> vptr Student::work()
};

int main(){
    Person *p = new Student();  // upcast

    p->work();      // student

    return 0;
}
```



Virtual destructor

- When we delete the derived class object using base class pointer -> only base class destructor will be called
- To call both base and derived class destructor -> base class destructor should be virtual
- If base class destructor is not virtual -> only base class destructor called -> resource leak
- If base class destructor is virtual -> derived class destructor called first -> then base class destructor called
- to free the resources properly

```
class Person{
public:
    string name;
    int age;

    virtual void work(){
        cout << "Person working..." << endl;
    }

    virtual ~ Person(){     // virtual destructor
        cout << "Person Destructor..." << endl;
    }
};

class Student : public Person{
public:
    string courseName;
```

```
    void work()  {
        cout << "Doing study..." <<endl;
    }

    ~ Student(){
        cout << "Student Destructor..." << endl;
    }
};

int main(){

    Person *p = new Student();  // upcast

    p->work();      // student

    delete p;       // to call both destructors

    return 0;
}
```

Output if base class destructor is not virtual:

```
Doing study...
Person Destructor...
```

Output if base class destructor is virtual:

```
Doing study...
Student Destructor...
Person Destructor...
```

RTTI

- RunTime Type Information
- To know the type of object at runtime
- typeid operator -> to get the type information of object at runtime
- It returns the reference of type_info class object
- type_info class is defined in `<typeinfo>` header file
- typeid(object).name() -> to get the name of the type of object

```
int main(){

    Person *p = new Student();
    Student *s = new Student();
```

```
    cout << typeid(*p).name() << endl;
    cout << typeid(*s).name() << endl;

    if(typeid(*p) == typeid(Student)){
        cout << "equal" << endl;
    }
    return 0;
}
```

Example:

```
class Person{
public:
    string name;
    int age;

    virtual void work(){
        cout << "Person working..." << endl;
    }

    virtual ~ Person(){
        cout << "Person Destructor..." << endl;
    }
};

class Student : public Person{
public:
    string courseName;

    void work()  {
        cout << "Doing study..." <<endl;
    }

    ~ Student(){
        cout << "Student Destructor..." << endl;
    }
};

int main(){

    //downcasting
//  Person *p = new Person(); //valid
//  Student *s = dynamic_cast<Person*>(p);  // safer -> downcasting error : not
valid

    Student *s = new Student();
//  Person *p = dynamic_cast<Person*>(p);

    if(Person *p = dynamic_cast<Student*>(s)){
        cout << "in if student -> dynamic cast in person"
    }
```

```
//  p->work();
//  s->work();
//
//  cout << typeid(*p).name() << endl;
//  cout << typeid(*s).name() << endl;

//  delete p;
//  delete s;

    return 0;
}
```

Advanced casts

- C++ provides 4 types of cast operators for type conversion
- C-style cast -> (type)object
- C++ style cast -> 4 types

dynamic_cast

- safely upcast the object at runtime
- used for runtime polymorphism
- used for upcasting and downcasting mostly upcasting

```
    Person *p = new Student();  // upcast

    Student *s = dynamic_cast<Student*>(p); // safer -> downcasting
```

```
    Student *s = new Student(); // upcast
    Person *p = dynamic_cast<Person*>(s);   // safer -> upcasting
```

static_cast

- compile time type coversion
- used for converting basic data types
- used for upcasting and downcasting

```
    Person *p = new Student();  // upcast

    Student *s = static_cast<Student*>(p);  // unsafe -> downcasting
```

```

```

```
    Student *s = new Student(); // upcast
    Person *p = static_cast<Person*>(s);    // safe -> upcasting
```

const_cast

- to change const <-> non-const
- used to add or remove constness of object

```
    const int a = 10;
    int *b = const_cast<int*>(&a);  // remove constness

    *b = 20;    // undefined behaviour

    cout << a << endl;  // 10
    cout << *b << endl; // 20
```

```
void print(char *s){
    cout << s << endl; //s is non-const : to be safe
}

void printSafe(const char *s){
    print(const_cast<char*>(s));
    // remove constness -> unsafe: modifying const object
}

int main() {
    const string str = "shil";
    printSafe(str);

    return 0;
}
```

reinterpret_cast

- raw memory bit manipulation -> hardware level, os -> code most of not used

```
    long ptr = 123456;
    int *p = reinterpret_cast<int*>(ptr);   // raw memory manipulation

    cout << p << endl;
```

output:

```
0x1e240
```

```
    int a = 65;
    char c = reinterpret_cast<char&>(a);    // raw memory manipulation

    cout << c << endl;
```

output:

```
A
```

```
    int a = 0x13984456;
    char *c = reinterpret_cast<char*>(&a);  // raw memory manipulation

    cout << *c << endl;
```

output:

```
V // depending on little/big endian
```

> big-endian -> it stores the most significant byte at the smallest memory address

> little-endian -> it stores the least significant byte at the smallest memory address

Pure virtual function

- A function which does not have definition in base class, and it is initiliazed with 0;
- It is declared using `virtual` keyword
- It is used to create abstract class

Syntax:

```
    virtual return_type function_name() = 0;
```

Abstract class

- A class which has at least one pure virtual function
- If you create a pure virtual function in a class, it will become a abstract class.
- We cannot create object of the abstract class but we can create reference of it.

- We can inherit the abstract class and provide definition to the pure virtual function in derived class.

```
class Person{       //abstract class
public:
    string name;
    int age;

    virtual void work() = 0; // pure virtual function

    //allowed
    void details(){
        cout << name << age << endl;
    }

    virtual ~ Person(){
//      cout << "Person Destructor..." << endl;
    }
};

class Student : public Person{
public:
    string courseName;

    void work() override  {
        cout << "Doing study..." <<endl;
    }

    ~ Student(){
//      cout << "Student Destructor..." << endl;
    }
};

class Teacher : public Person{
public:
    string subject;

    void work() override {
        cout << "Teaching Students..." <<endl;
    }

    ~ Teacher(){
//      cout << "Teacher Destructor..." << endl;
    }
};

int main() {

//  Person p;       // person class object -> not valid :error
    Student s;
    Teacher t;
    Person &p = s;      // abstract class reference -> student class object

//  only references and pointers
```

```
//  value -> error
    p.work();
    s.work();
//  s.details();
    t.work();

    return 0;
}
```