

OOP

Lecture 24

18/01/2022

Using Template class in aggregation :

If we want to use a template class in the aggregation, we need to specify the data type of the object which is to be aggregated. The data type can be passed statically, or we can make the second class a template class as well. Both of these functionalities will work depending upon the situation. The simplest program to show is :

Specify data type :

```
template <class Type>
class A{
    Type x;
};

class B{
    A<int> objA;
public:
};
```

Making Template :

```
template <class Type>
class A{
    Type x;
};

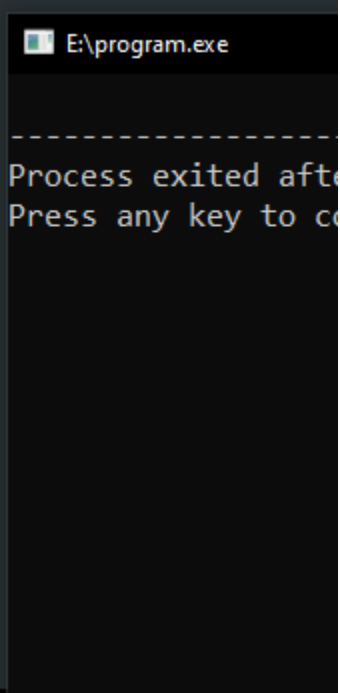
template <class T>
class B{
    A<T> objA;
};
```

Sample Program :

```
template <class Type>
class A{
    Type x;
public:
};

template <class T, class S>
class B{
    A<T> objA;
    S y;
public:
};

int main()
{
    B<char, char> objB1;
    B<int, char> objB2;
    B<int, float> objB3;
    return 0;
}
```



Using Class Template in Inheritance :

This concept also applies when we try to inherit the template class in the child object. Either we can specify primitive / built-in data type when inheriting or we can make the child class as template class as well and pass the template type to the parent class.

Specify data type :

```
template <class Type>
class A{
    Type x;
};

class B : public A<int>{

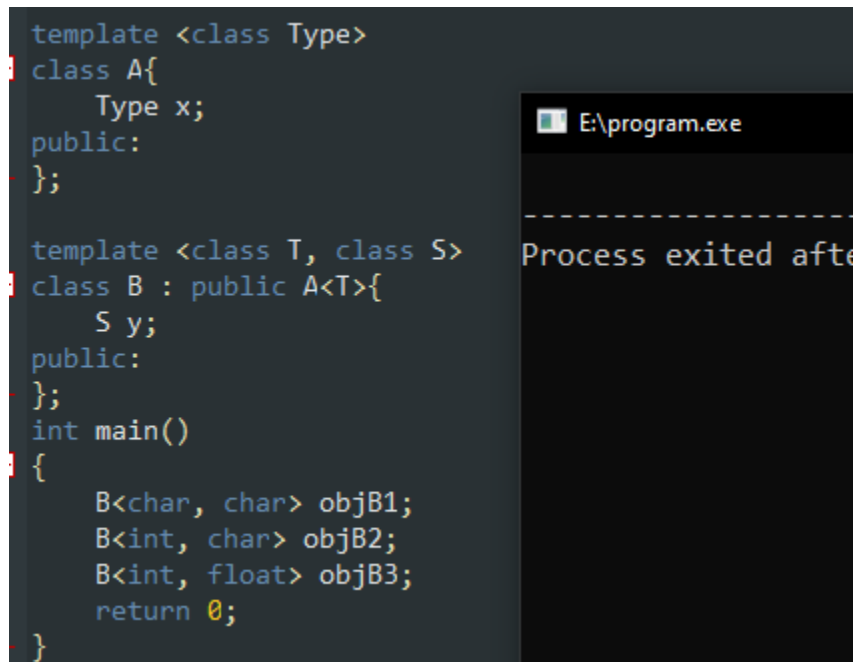
};
```

Making Template :

```
template <class Type>
class A{
    Type x;
};

template <class T>
class B : public A<T>{
};
```

Sample Program :



```
template <class Type>
class A{
    Type x;
public:
};

template <class T, class S>
class B : public A<T>{
    S y;
public:
};

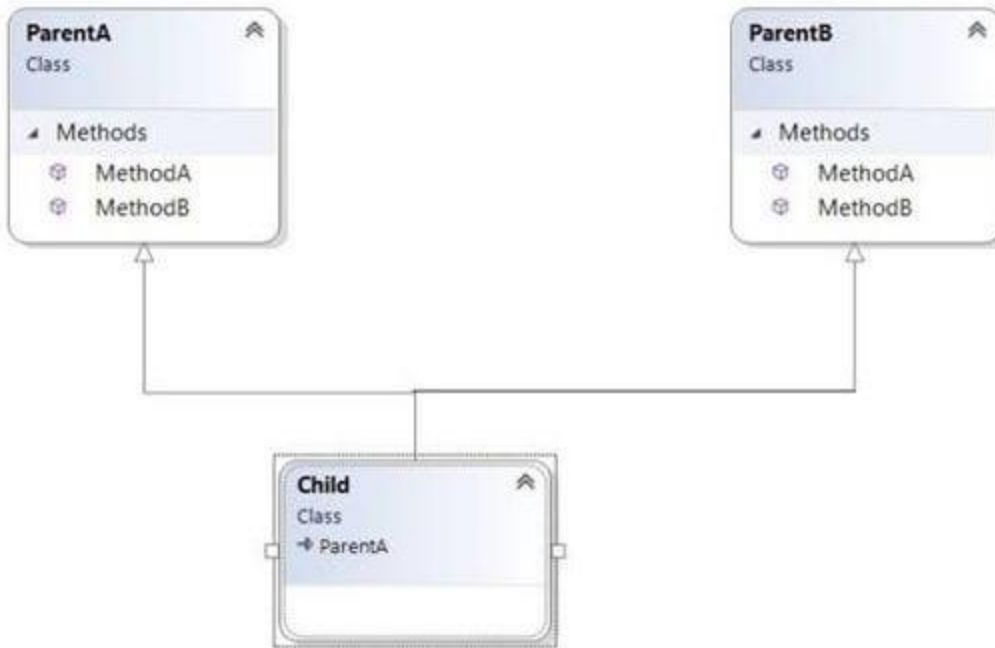
int main()
{
    B<char, char> objB1;
    B<int, char> objB2;
    B<int, float> objB3;
    return 0;
}
```

E:\program.exe

Process exited after

Multiple Inheritance :

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. In multiple Inheritance, we derive a class from more than one parent. The UML diagram of the multiple inheritance looks like this :



The child class has 2 parent classes, ParentA and ParentB.

The constructors of inherited classes are called in the same order in which they are inherited. It means that in the order they are written after colon, while defining the child class. For example, in the following program, B's constructor is called before A's constructor.

```

class A{
public:
    A(){    cout << "Constructor Class A\n";    }
    ~A(){   cout << "Destructor Class A\n";   }
};
class B{
public:
    B(){    cout << "Constructor Class B\n";    }
    ~B(){   cout << "Destructor Class B\n";   }
};
class C:public B, public A{
public:
    C(){    cout << "Constructor Class C\n";    }
    ~C(){   cout << "Destructor Class C\n";   }
};
int main(){
    C objC;
    return 0;
}
  
```

E:\program.exe

```

Constructor Class B
Constructor Class A
Constructor Class C
Destructor Class C
Destructor Class A
Destructor Class B
  
```

 Process exited after 2.32
 Press any key to continue

The destructors are called in reverse order of constructors.

Before going further, make sure you understand the concept of multiple inheritance. We are not going into details. Just understand that the child class C will inherit all the properties of class A and class B and it will have access to all the public and protected members of both of the classes.

```
class A{
public:
    A(){    cout << "Constructor Class A\n";    }
    ~A(){   cout << "Destructor Class A\n"; }
    void fA(){    cout << "Function Class A\n";    }
};
class B{
public:
    B(){    cout << "Constructor Class B\n";    }
    ~B(){   cout << "Destructor Class B\n"; }
    void fB(){    cout << "Function Class B\n";    }
};
class C:public B, public A{
public:
    C(){    cout << "Constructor Class C\n";    }
    ~C(){   cout << "Destructor Class C\n"; }
};
int main(){
    C objC;
    objC.fA();
    objC.fB();
    return 0;
}
```

```
E:\program.exe
Constructor Class B
Constructor Class A
Constructor Class C
Function Class A
Function Class B
Destructor Class C
Destructor Class A
Destructor Class B

-----
Process exited after 0.3248
Press any key to continue .
```

Problems Associated with Multiple Inheritance :

The most obvious problem with multiple inheritance occurs during function overloading. Suppose two parent classes have the same function which is not refined or overridden in the child class. If we try to call the function using the object of the child class, the compiler shows error. It is because the compiler doesn't know which function to call.

```
class A{
public:
    A(){    cout << "Constructor Class A\n";    }
    ~A(){   cout << "Destructor Class A\n"; }
    void fA(){    cout << "Function Class A\n";    }
    void commonFunction(){    cout << "Common Function Class A\n";    }
};
class B{
public:
    B(){    cout << "Constructor Class B\n";    }
    ~B(){   cout << "Destructor Class B\n"; }
    void fB(){    cout << "Function Class B\n";    }
    void commonFunction(){    cout << "Common Function Class B\n";    }
};
class C:public B, public A{
public:
    C(){    cout << "Constructor Class C\n";    }
    ~C(){   cout << "Destructor Class C\n"; }
};
int main(){
    C objC;
    objC.fA();
    objC.fB();
    objC.commonFunction();
    return 0;
}
```

Resources Compile Log Debug Find Results X Close

	Message
am.cpp	In function 'int main()':
am.cpp	[Error] request for member 'commonFunction' is ambiguous
am.cpp	[Note] candidates are: void A::commonFunction()

Solutions : There are 2 solutions for this problem,

1. We can specify the class from which the function should be called by using scope resolution operator (::).
2. We can redefine this function in the child class and call the parent class functions explicitly there.

Method 1 :

```
class A{
public:
    A(){    cout << "Constructor Class A\n";    }
    ~A(){   cout << "Destructor Class A\n"; }
    void fA(){    cout << "Function Class A\n";    }
    void commonFunction(){    cout << "Common Function Class A\n";    }
};
class B{
public:
    B(){    cout << "Constructor Class B\n";    }
    ~B(){   cout << "Destructor Class B\n"; }
    void fB(){    cout << "Function Class B\n";    }
    void commonFunction(){    cout << "Common Function Class B\n";    }
};
class C:public B, public A{
public:
    C(){    cout << "Constructor Class C\n";    }
    ~C(){   cout << "Destructor Class C\n"; }
};
int main(){
    C objC;
    objC.A::commonFunction();
    objC.B::commonFunction();
    return 0;
}
```

E:\program.exe

Constructor Class B
Constructor Class A
Constructor Class C
Common Function Class A
Common Function Class B
Destructor Class C
Destructor Class A
Destructor Class B

Process exited after 5.74 s
Press any key to continue .

Method 2 :

```
class A{
public:
    A(){    cout << "Constructor Class A\n";    }
    ~A(){   cout << "Destructor Class A\n"; }
    void fA(){    cout << "Function Class A\n";    }
    void commonFunction(){    cout << "Common Function Class A\n";    }
};
class B{
public:
    B(){    cout << "Constructor Class B\n";    }
    ~B(){   cout << "Destructor Class B\n"; }
    void fB(){    cout << "Function Class B\n";    }
    void commonFunction(){    cout << "Common Function Class B\n";    }
};
class C:public B, public A{
public:
    C(){    cout << "Constructor Class C\n";    }
    void commonFunction(){    // Redefining
        A::commonFunction();
        B::commonFunction();
        cout << "Common Function Class C\n";
    }
    ~C(){   cout << "Destructor Class C\n"; }
};
int main(){
    C objC;
    objC.commonFunction();
    return 0;
}
```

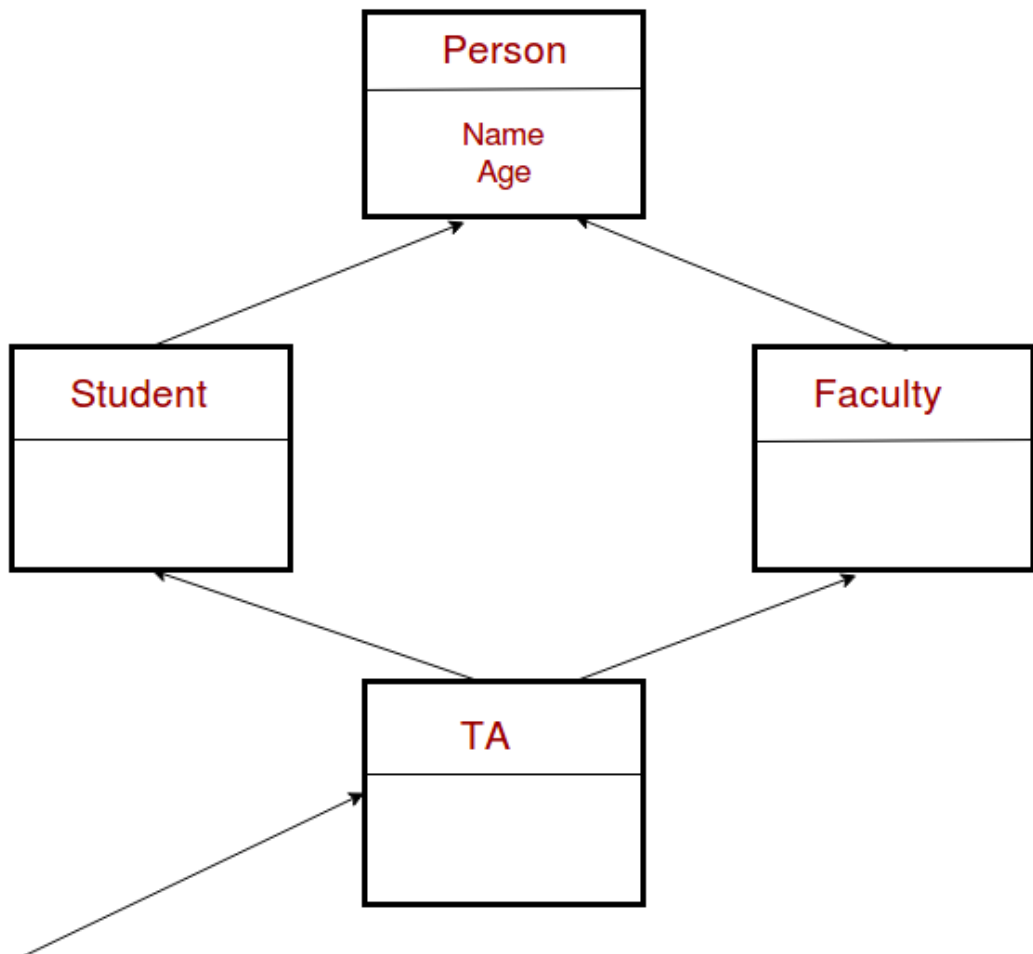
E:\program.exe

Constructor Class B
Constructor Class A
Constructor Class C
Common Function Class A
Common Function Class B
Common Function Class C
Destructor Class C
Destructor Class A
Destructor Class B

Process exited after 0.6114 sec
Press any key to continue .

The Diamond Problem :

The diamond problem occurs when two parent classes of a child class have a common parent class. For example, check out this diagram,



Name and Age needed only once

The members of the parent class “Person” are inherited by Student and Faculty classes. And then the TA class inherits both of these classes. It means that the TA class gets two copies of all members (data members and member functions) of Person class. It creates ambiguities for the compiler.

Now, we are writing the code of the diamond problem to first show the constructor calling sequence of such inheritance.


```
#include<iostream>
using namespace std;
class Person {
public:
    Person(){ cout << "Person class constructor called" << endl; }
};

class Faculty : public Person {
public:
    Faculty(){ cout<<"Faculty class constructor called"<< endl; }
};

class Student : public Person {
public:
    Student(){ cout<<"Student class constructor called"<< endl; }
};

class TA : public Faculty, public Student {
public:
    TA(){
        cout<<"TA class constructor called"<< endl;
    }
};

int main() {
    TA ta1;
}
```

Et\program.exe

Person class constructor called
Faculty class constructor called
Person class constructor called
Student class constructor called
TA class constructor called

Process exited after 1.144 seconds with r
Press any key to continue . . .

As we can see here the constructor of the 'Person' class is called two times. The destructor of the 'Person' class will also be called two times when the object 'ta1' is destructed. So object 'ta1' has two copies of all members(data members and member functions) of the 'Person' class, it is because of two parent classes, as each parent class will inherit the Person class separately, this causes ambiguities.

The solution to this problem is the **virtual** keyword. We make the classes **Faculty** and **Student** as virtual base classes to avoid two copies of 'Person' in 'TA' class. It means that while inheriting the Person class in Faculty and Student classes we will write a virtual keyword before public, it will look like this,

```
class Faculty : virtual public Person { }  
class Student : virtual public Person{ }
```

Now, when we inherit Faculty and Student classes in the TA class, the compiler will check if the Person class is already used then it will not create an instance of the Person class again. The sample program after

writing virtual keywords is as follows,

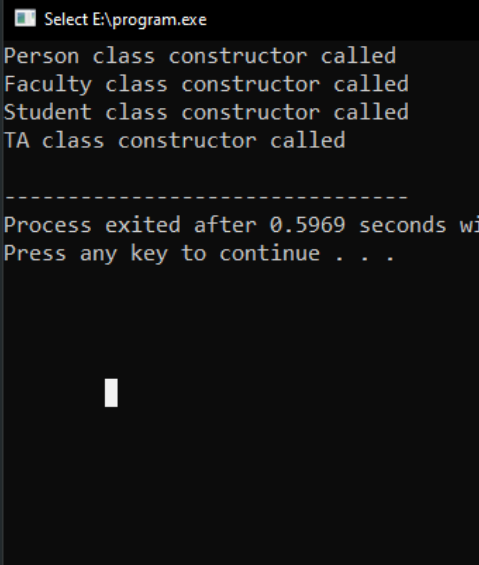
```
#include<iostream>
using namespace std;
class Person {
public:
    Person(){ cout << "Person class constructor called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(){ cout<<"Faculty class constructor called"<< endl; }
};

class Student : virtual public Person {
public:
    Student(){ cout<<"Student class constructor called"<< endl; }
};

class TA : public Faculty, public Student {
public:
    TA(){
        cout<<"TA class constructor called"<< endl;
    }
};

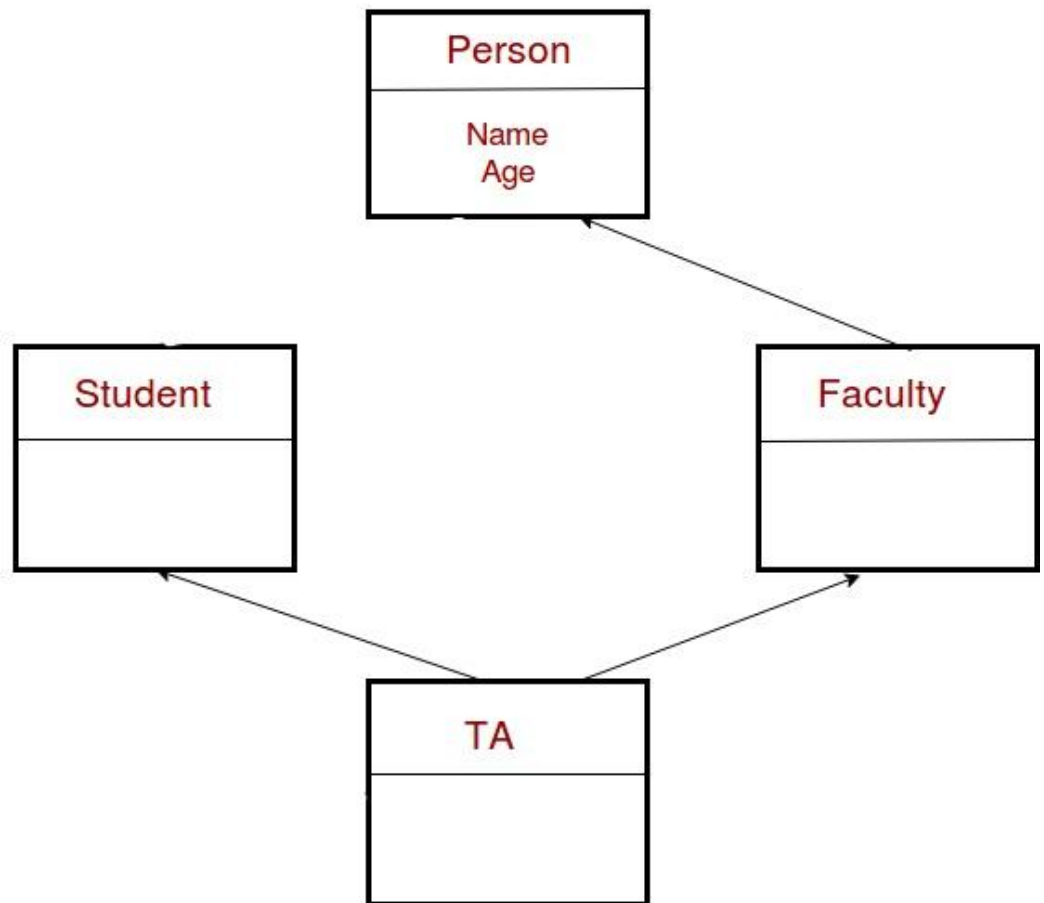
int main() {
    TA ta1;
}
```



```
Select E:\program.exe
Person class constructor called
Faculty class constructor called
Student class constructor called
TA class constructor called

-----
Process exited after 0.5969 seconds with return code 0
Press any key to continue . . .
```

Now we can see that the Person class constructor is called only once and the inheritance hierarchy will like this now.



Noticed the change? The Student class is not connected with the Person class, instead the members of the Person class will only be derived from the Faculty class. It is because the constructor of the Faculty class is called first (stated earlier that the constructor of which parent class is called first), So, the Faculty class will call the constructor of Person class. But, when the constructor of Student class is called it will not call the Person class constructor again or in other words it will not create the instance of Person class again because of the **virtual** keyword.

Typecasting in C++ :

Typecasting is also called as type conversion. It means converting one data type into another. There are two types of type conversion: implicit and explicit type conversion,

- Implicit type conversion operates automatically when the compatible data type is found. The compiler does implicit type conversion on its own. The lower data type is always converted to the superior data type. Like, char is converted to int, int is converted to float etc but not vice versa, when required.

Example :

Using an int and float in an expression like $a + b$ where a is int and b is float. So, in this case a will be converted to float.

- The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.

Syntax :

(type) variable;

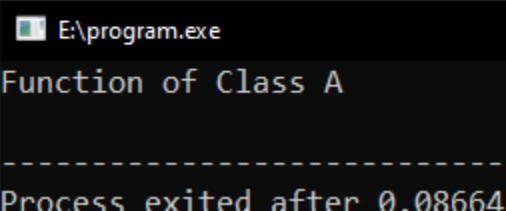
It can be used in any expression.

Like, (float)a / b; where a is an integer

Type Casting in inheritance :

When we do polymorphism, we create a parent class pointer and child class object. Just to recall check out this program,

```
class A{
public:
    void fA(){ cout << "Function of Class A\n"; }
};
class B:public A{
public:
    void fB(){ cout << "Function of Class B\n"; }
};
int main(){
    A *ptrA = new B;
    ptrA->fA();
    //ptrA->fB();
    return 0;
}
```



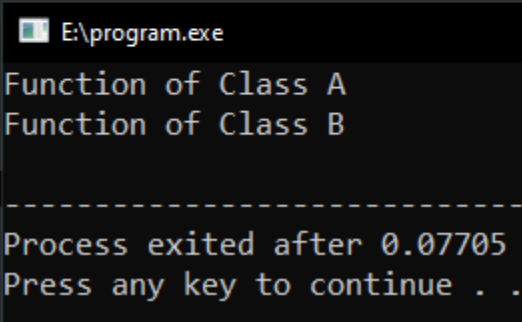
The pointer can call the functions of the parent class but it cannot call the functions of the child class directly, (if not overridden). If we try to uncomment the line and try to run, it will generate an error.

Now, just out of curiosity we made a pointer of child class and tried to assign the parent class pointer to child class pointer, but the compiler again generated an error.

```
class A{
public:
    void fA(){ cout << "Function of Class A\n"; }
};
class B:public A{
public:
    void fB(){ cout << "Function of Class B\n"; }
};
int main(){
    A *ptrA = new B;
    ptrA->fA();
    B *ptrB = ptrA;
    ptrB->fB();
    return 0;
}
```

Even though the parent class pointer is pointing to a child class object, the compiler does not know this. It only knows that we are trying to assign the parent pointer to the child pointer and it is not allowed. Why is it not allowed? We will discuss this at the end of this lecture. Here, we know that the parent class pointer is pointing to the child class object so we can typecast the pointer into the child class type explicitly, to make the assignment valid. Here's the code which is now working fine.

```
class A{
public:
    void fA(){ cout << "Function of Class A\n"; }
};
class B:public A{
public:
    void fB(){ cout << "Function of Class B\n"; }
};
int main(){
    A *ptrA = new B;
    ptrA->fA();
    B *ptrB = (B*)ptrA;
    ptrB->fB();
    delete ptrB;
    return 0;
}
```



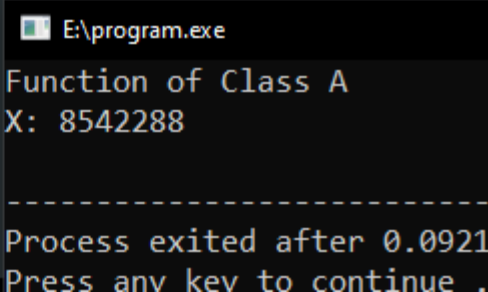
Now, the ptrB can call the members of class B as well without even overriding the function.

There is one problem that may occur when we try to do this type of typecasting. Consider a case when we made a parent class pointer and parent class object which is completely fine and normal, not a big deal. But, if we type cast the parent pointer and assigned it to the child pointer and started calling the child class members, what will happen, Check out this code,

```

class A{
public:
    void fA(){ cout << "Function of Class A\n"; }
};
class B:public A{
    int x;
public:
    B() { x = 10; }
    void fB(){ cout << "X: " << x << "\n"; }
};
int main()
{
    A *ptrA = new A;
    ptrA->fA();
    B *ptrB = (B*)ptrA;
    ptrB->fB();
    return 0;
}

```



```

E:\program.exe
Function of Class A
X: 8542288
-----
Process exited after 0.0921
Press any key to continue .

```

Here, we can see the error. The compiler didn't give any error because it warned us before type casting and gave an error there. But, we took the risk and said that we know what we are doing and explicitly type cast the parent pointer to the child pointer, and now see the consequences. Even though the child pointer now has access to the members of the child class, but the child class object hasn't been created yet. So, the constructor of the child class hasn't been called and the x contains the garbage value, which is printed on the screen. So that's the reason why we should avoid type casting the parent pointer to child pointer.

-THE END-