Part: 31

Constructor Overloading In C++

In this tutorial, we will discuss constructor overloading in C++

Constructor Overloading in C++

Constructor overloading is a concept in which one class can have multiple constructors with different parameters. The main thing to note here is that the constructors will run according to the arguments for example if a program consists of 3 constructors with 0, 1, and 2 arguments, so if we pass 1 argument to the constructor the compiler will automatically run the constructor which is taking 1 argument. An example program to demonstrate the concept of Constructor overloading in C++ is shown below.

```
#include <iostream>
using namespace std;
class Complex{
    int a, b;
public:
    Complex(){
        a = 0;
        b =0;
    }
    Complex(int x, int y){
        a = x;
        b = y;
    }
    Complex(int x){
        a = x;
        b = 0;
    }
    void printNumber(){
        cout << "Your number is " << a << " + " << b << "i" <<
endl;
    }
};
```

Code Snippet 1: Constructor Overloading Program Example

As shown in Code Snippet 1,

- 1st we created a **"complex"** class which consists of private data members **"a"** and **"b"**.

- 2nd default constructor of the **"complex"** class is declared which has no parameters and assigns **"0"** to the data members **"a"** and **"b"**.
- 3rd parameterized constructor of the **"complex"** class is declared which takes two parameters and assigns values to the data members **"a"** and **"b"**.
- 4th parameterized constructor of the **"complex"** class is declared which takes one parameter and assigns values to the data members **"a"** and **"b"**.
- 5th function **"printNumber"** is defined which will print the values of the data members **"a"** and **"b"**.

The main program is shown in code snippet 2.

```
int main()
{
    Complex c1(4, 6);
    c1.printNumber();

    Complex c2(5);
    c2.printNumber();

    Complex c3;
    c3.printNumber();
    return 0;
}
```

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st parameterized constructor is called with the object **"c1"** and the values **"4"** and **"6"** are passed. The main thing to note here is that this will run the constructor with two parameters.
- 2nd function **"printNumber"** is called which will print the values of data members
- 3rd parameterized constructor is called with the object **"c2"** and the value **"5"** is passed. The main thing to note here is that this will run the constructor with one parameter.
- 4th function **"printNumber"** is called which will print the values of data members

- 5th default constructor is called with the object **"c3"**. The main thing to note here is that this will run the constructor with no parameters.
- 6th function **"printNumber"** is called which will print the values of data members

The output for the following program is shown in figure 1.

```
Your number is 4 + 6i
Your number is 5 + 0i
Your number is 0 + 0i
```

Figure 1: Program Output

As shown in figure 1, all the values which were passed and assigned through parameterized constructors and the values which were assigned through the default constructor are printed.

Part: 32

Constructors With Default Arguments In C++

In this tutorial, we will discuss constructors with default arguments in C++

Constructors with Default Arguments in C++

Default arguments of the constructor are those which are provided in the constructor declaration. If the values are not provided when calling the constructor, the constructor uses the default arguments automatically. An example program to demonstrate the concept default arguments in C++ is shown below.

```
#include<iostream>
using namespace std;
class Simple{
    int data1;
    int data2;
    int data3;
    public:
        Simple(int a, int b=9, int c=8){
            data1 = a;
            data2 = b;
            data3 = c;
        }

       void printData();
};
void Simple :: printData(){
    cout<<"The value of data1, data2 and data3 is "<<data1<<", "<<
data2<<" and "<< data3<<endl;
}
```

Code Snippet 1: Constructor with Default Arguments Program Example

As shown in a code snippet 1,

- 1st we created a **“simple”** class which consists of private data members **“data1”**, **“data2”** and **“data3”**.
- 2nd parameterized constructor of the **“simple”** class is defined which takes three parameters and assigns values to the data members **“a”** and **“b”**. The main thing to note here is that the value **“9”** and **“8”** are the default values for the variables **“b”** and **“c”**.

- 3rd function **"printData"** is defined which prints the values of the data members **"data1"**, **"data2"**, and **"data3"**.

The main program is shown in code snippet 2.

```
int main(){
    Simple s(12, 13);
    s.printData();
    return 0;
}
```

Code Snippet 2: Main Program

As shown in code snippet 2,

- 1st parameterized constructor is called with the object **"s"** of the data type **"simple"** and the values **"12"** and **"13"** are passed. The main thing to note here is that the value of the parameter **"c"** will be automatically set by the default value.
- 2nd function **"printData"** is called which will print the values of data members.

The output for the following program is shown in figure 1.

```
PS D:\MyData\Business\code playground\C++ course> cd "d:\MyI
The value of data1, data2 and data3 is 12, 13 and 8
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Program Output

As shown in figure 1, the value **"12"**, **"13"**, and **"8"** are printed. The constructor assigned the values **"12"** and **"13"** to the variables **"a"** and **"b"** but the value for the variable **"c"** was not passed that's why constructors set the value **"8"** which was the default value for the variable **"c"**.

Part: 33

Dynamic Initialization of Objects Using Constructors

In this tutorial, we will discuss the dynamic initialization of objects using constructors in C++

Dynamic Initialization of Objects Using Constructors

The dynamic initialization of the object means that the object is initialized at the runtime. Dynamic initialization of the object using a constructor is beneficial when the data is of different formats. An example program is shown below to demonstrate the concept of dynamic initialization of objects using constructors.

```
#include<iostream>
using namespace std;
class BankDeposit{
    int principal;
    int years;
    float interestRate;
    float returnValue;
    public:
       BankDeposit(){}
       BankDeposit(int p, int y, float r); // r can be a value like 0.04
       BankDeposit(int p, int y, int r); // r can be a value like 14
       void show();
};
```

Code Snippet 1: Dynamic Initialization of Objects using Constructor Example

As shown in Code Snippet 1,

- 1st we created a **"BankDeposit"** class which consists of private data members **"principal"**, **"years"**, **"interestRate"**, and **"returnValue"**.
- 2nd default constructor of the **"BankDeposit"** class is declared.
- 3rd parameterized constructor of the **"BankDeposit"** class is declared which takes three parameters **"p"**, **"y"**, and **"r"**. The main thing to note here is that the parameter **"r"** is of a float data type.
- 4th parameterized constructor of the **"BankDeposit"** class is declared which takes three parameters **"p"**, **"y"**, and **"r"**. The main thing to note here is that the parameter **"r"** is of an integer data type.

- 5th function **"show"** is declared.

The definition of constructors and function is shown below.

```
BankDeposit :: BankDeposit(int p, int y, float r){
    principal = p;
    years = y;
    interestRate = r;
    returnValue = principal;
    for (int i = 0; i < y; i++){
        returnValue = returnValue * (1+interestRate);
    }
}
BankDeposit :: BankDeposit(int p, int y, int r){
    principal = p;
    years = y;
    interestRate = float(r)/100;
    returnValue = principal;
    for (int i = 0; i < y; i++){
        returnValue = returnValue * (1+interestRate);
    }
}

void BankDeposit :: show(){
    cout<<endl<<"Principal amount was "<<principal
        << ". Return value after "<<years
        << " years is "<<returnValue<<endl;
}
```

Code Snippet 2: Definition of Constructors and Function

As shown in Code snippet 2,

- 1st the constructor **"BankDeposit"** is defined in which the value of the parameter **"p"** is assigned to the data member **"principal"**; the value of the parameter **"y"** is assigned to the data member **"year"**; the value of the parameter **"r"** is assigned to the data member **"interestRate"**. At the end **"for"** loop is defined which will run till the length of the variable **"y"** and add **"1"** in the **"interestRate"**; then multiply the value with the **"returnValue"**. The main thing to note here is that in this constructor the data type of the parameter **"r"** is float.
- 2nd another constructor **"BankDeposit"** is defined in which the value of the parameter **"p"** is assigned to the data member **"principal"**; the value of the parameter **"y"** is assigned to the data member **"year"**; the value of the

parameter **"r"** is converted to **"float"** and divided by **"100"** then assigned to the data member **"interestRate"**. At the end **"for"** loop is defined which will run till the length of the variable **"y"** and add **"1"** in the **"interestRate"**; then multiply the value with the **"returnValue"**. The main thing to note here is that in this constructor the data type of the parameter **"r"** is float.

- 3rd the function **"show"** is defined which will print the values of the data members **"principal"**, **"year"**, and **"returnValue"**.

The main program is shown in code snippet 3.

```
int main(){
    BankDeposit bd1, bd2, bd3;
    int p, y;
    float r;
    int R;
    cout<<"Enter the value of p y and r"<<endl;
    cin>>p>>y>>r;
    bd1 = BankDeposit(p, y, r);
    bd1.show();

    cout<<"Enter the value of p y and R"<<endl;
    cin>>p>>y>>R;
    bd2 = BankDeposit(p, y, R);
    bd2.show();
    return 0;
}
```

Code Snippet 3: Main Program

As shown in a code snippet 3,

- 1st the object **"bd1"**, **"bd2"**, and **"bd3"** of the data type **"BankDeposit"** are created.
- 2nd the integer variables **"p"** and **"y"** are declared; the float variable **"r"** is declared, and the integer variable **"R"** is declared.
- 3rd the values for the variables **"p"**, **"y"**, and**"r"** are taken from the user on the runtime.
- 4th parameterized constructor **"BankDeposit"** is called with the object **"bd1"** and the variables **"p"**, **"y"**, and **"r"** are passed. The main thing to note here is that this will run the constructor with float parameters **"r"**.

- 5th function "show" is called which will print the values of data members
- 6th the values for the variable's **"p"**, **"y"**, and **"R"** are taken from the user on the runtime.
- 7th parameterized constructor **"BankDeposit"** is called with the object **"bd2"** and the variables **"p"**, **"y"**, and **"R"** are passed. The main thing to note here is that this will run the constructor with integer parameters **"R"**.
- 8th function **"show"** is called which will print the values of data members.

The output for the following program is shown in figure 1.

```
Enter the value of p y and r
100
1
0.05

Principal amount was 100. Return value after 1 years is 105
Enter the value of p y and R
100
1
5

Principal amount was 100. Return value after 1 years is 105
```

Figure 1: Program Output

As shown in figure 1, the first time the values **"100"**, **"1"**, and **"0.05"** are entered and it gives us the return value of **"105"**. The second time the values **"100"**, **"1"**, and **"5"** are entered and it gives us the return value of **"105"**. So, the main thing to note here is that the compiler figures out the run time by seeing the data type and runs the relevant constructor.

Part: 34

Copy Constructor in C++

In this tutorial, we will discuss copy constructor in C++

Copy Constructor in C++

A copy constructor is a type of constructor that creates a copy of another object. If we want one object to resemble another object we can use a copy constructor. If no copy constructor is written in the program compiler will supply its own copy constructor. An example program to demonstrate the concept of a Copy constructor in C++ is shown below.

```
#include<iostream>
using namespace std;
class Number{
    int a;
    public:
        Number(){
            a = 0;
        }
        Number(int num){
            a = num;
        }
// When no copy constructor is found, compiler supplies its own copy constructor
        Number(Number &obj){
            cout<<"Copy constructor called!!!"<<endl;
            a = obj.a;
        }
        void display(){
            cout<<"The number for this object is "<< a <<endl;
        }
};
```

Code Snippet 1: Copy Constructor Example Program

As shown in Code Snippet 1,

- 1st we created a **"number"** class which consists of private data member **"a"**.
- 2nd default constructor of the **"number"** class is defined which has no parameters and assign **"0"** to the data members **"a"**.
- 3rd parameterized constructor of the **"number"** class is defined which takes one parameter and assigns values to the data members **"a"**.

- 4th copy constructor of the **"number"** class is defined which takes its own reference object as a parameter and assigns values to the data members **"a"**.
- 5th function **"display"** is defined which will print the values of the data members **"a"**.

The main program is shown in code snippet 2.

```
int main(){
    Number x, y, z(45), z2;
    x.display();
    y.display();
    z.display();
    Number z1(z); // Copy constructor invoked
    z1.display();
    z2 = z; // Copy constructor not called
    z2.display();
    Number z3 = z; // Copy constructor invoked
    z3.display();
    // z1 should exactly resemble z  or x or y
    return 0;
}
```

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st objects **"x"**, **"y"**, **"z"**, and **"z1"** are created of the **"number"** data type. The main thing to note here is that the object **"z"** has a value **"45"**.
- 2nd function **"display"** is called by the object's **"x"**, **"y"**, and **"z"**.
- 3rd copy constructor is invoked and the object **"z"** is passed to **"z1"**
- 4th function **"display"** is called by the object **"z1"**
- 5th the value of **"z"** is assigned to **"z1"**. The main thing to note here is that it will not invoke a copy constructor because the object **"z"** is already created.
- 6th function **"display"** is called by the object **"z2"**
- 7th the value of **"z"** is assigned to **"z3"**. The main thing to note here is that it will invoke a copy constructor because the object **"z3"** is being created.
- 8th function **"display"** is called by the object **"z3"**

The output for the following program is shown in figure 1.

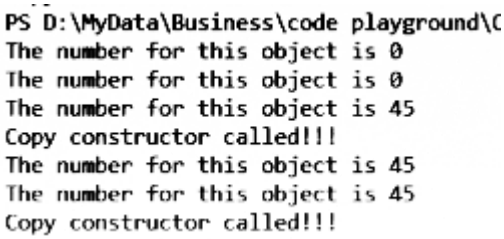

Figure 1: Program Output

As shown in figure 1, all the values which were passed and assigned through copy constructors are printed.

Part: 35

Destructor in C++

In this tutorial, we will discuss Destructor in C++

Destructor in C++

A destructor is a type of function which is called when the object is destroyed. Destructor never takes an argument nor does it return any value. An example program to demonstrate the concept of destructors in C++ is shown below.

```
#include<iostream>
using namespace std;
// Destructor never takes an argument nor does it return any value
int count=0;
class num{
    public:
        num(){
            count++;
            cout<<"This is the time when constructor is called for
object number"<<count<<endl;
        }
        ~num(){
            cout<<"This is the time when my destructor is called
for object number"<<count<<endl;
            count--;
        }
};
```

Code Snippet 1: Destructor Example Program

As shown in Code Snippet 1,

- 1st global variable **"count"** is initialized.
- 2nd we created a **"num"** class.
- 3rd default constructor of the **"num"** class is defined which has no parameters and does increment in the variable **"count"** and prints its value. The main thing to note here is that every time the new object will be created this constructor will run.
- 4th destructor of the **"num"** class is defined. The destructor prints the value of the variable **"count"** and decrement in the value of **"count"**. The main thing

to note here is that every time the object has been destroyed this destructor will run.

The main program is shown in code snippet 2.

```
int main(){
    cout<<"We are inside our main function"<<endl;
    cout<<"Creating first object n1"<<endl;
    num n1;
    {
        cout<<"Entering this block"<<endl;
        cout<<"Creating two more objects"<<endl;
        num n2, n3;
        cout<<"Exiting this block"<<endl;
    }
    cout<<"Back to main"<<endl;
    return 0;
}
```

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st object **"n1"** is created of the **"num"** data type. The main thing to note here is that when the object **"n1"** is created the constructor will run.
- 2nd inside the block two objects **"n2"** and **"n3"** are created of the **"num"** data type. The main things to note here are that when the objects **"n2"** and **"n3"** are created the constructor will run for both objects and when the block ends the destructor will run for both objects **"n2"** and **"n3"**.
- 3rd when the program ends the destructor for the object **"n1"** will run.

The output for the following program is shown in figure 1.

```
Creating first object n1
This is the time when constructor is called for object number1
Entering this block
Creating two more objects
This is the time when constructor is called for object number2
This is the time when constructor is called for object number3
Exiting this block
This is the time when my destructor is called for object number3
This is the time when my destructor is called for object number2
Back to main
This is the time when my destructor is called for object number1
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Program Output

As shown in figure 1, first the constructor for the object **"n1"** was called; second the constructor for the objects **"n2"** and **"n3"** was called; third the destructor was called for the objects **"n2"** and **"n3"**; at the end destructor for the object **"n1"** was called.

Part: 36

Inheritance & Its Different Types with Examples in C++

In this tutorial, we will discuss inheritance in C++

Inheritance in C++ an Overview

- Reusability is a very important feature of OOPs
- In C++ we can reuse a class and add additional features to it
- Reusing classes saves time and money
- Reusing already tested and debugged classes will save a lot of effort of developing and debugging the same thing again

What is Inheritance in C++?

- The concept of reusability in C++ is supported using inheritance
- We can reuse the properties of an existing class by inheriting it
- The existing class is called a base class
- The new class which is inherited from the base class is called a derived class
- Reusing classes saves time and money
- There are different types of inheritance in C++

Forms of Inheritance in C++

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Single Inheritance in C++

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class. For example, we have two classes **“employee”** and **“programmer”**. If the **“programmer”** class is inherited from the **“employee”** class which means that the **“programmer”** class can now implement the functionalities of the **“employee”** class.

Multiple Inheritances in C++

Multiple inheritances are a type of inheritance in which one derived class is inherited with more than one base class. For example, we have three classes **"employee"**, **"assistant"** and **"programmer"**. If the **"programmer"** class is inherited from the **"employee"** and **"assistant"** class which means that the **"programmer"** class can now implement the functionalities of the **"employee"** and **"assistant"** class.

Hierarchical Inheritance

A hierarchical inheritance is a type of inheritance in which several derived classes are inherited from a single base class. For example, we have three classes **"employee"**, **"manager"** and **"programmer"**. If the **"programmer"** and **"manager"** classes are inherited from the **"employee"** class which means that the **"programmer"** and **"manager"** class can now implement the functionalities of the **"employee"** class.

Multilevel Inheritance in C++

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class. For example, we have three classes **"animal"**, **"mammal"** and **"cow"**. If the **"mammal"** class is inherited from the **"animal"** class and **"cow"** class is inherited from **"mammal"** which means that the "mammal" class can now implement the functionalities of **"animal"** and **"cow"** class can now implement the functionalities of **"mammal"** class.

Hybrid Inheritance in C++

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. In hybrid inheritance, a class is derived from two classes as in multiple inheritances. However, one of the parent classes is not a base class. For example, we have four classes **"animal"**, **"mammal"**, **"bird"**, and **"bat"**. If **"mammal"** and **"bird"** classes are inherited from the **"animal"** class and **"bat"** class is inherited from **"mammal"** and **"bird"** classes which means that **"mammal"** and **"bird"** classes can now implement the functionalities of **"animal"** class and "bat" class can now implement the functionalities of **"mammal"** and **"bird"** classes.

Part: 37

Inheritance Syntax & Visibility Mode in C++

In this tutorial, we will discuss inheritance syntax and visibility mode in C++

Inheritance Syntax and Visibility mode in C++

Inheritance is a process of inheriting attributes of the base class by a derived class. The syntax of the derived class is shown below.

```
 // Derived Class syntax
class {{derived-class-name}} : {{visibility-mode}} {{base-class-name}}
{
    class members/methods/etc...
}
```

Code Snippet 1: Derived Class syntax

As shown in a code snippet 1,

- After writing the class keyword we have to write the derived class name and then put a **“:”** sign.
- After **“:”** sign we have to write the visibility mode and then write the base class name.

Note:

- Default visibility mode is private
- Public Visibility Mode: Public members of the base class becomes Public members of the derived class
- Private Visibility Mode: Public members of the base class become private members of the derived class
- Private members are never inherited

An example program is shown below to demonstrate the concept of inheritance.

```
#include <iostream>
using namespace std;
// Base Class
class Employee
{
public:
    int id;
    float salary;
```

```
    Employee(int inpId)
    {
        id = inpId;
        salary = 34.0;
    }
    Employee() {}
};
// Creating a Programmer class derived from Employee Base class
class Programmer : public Employee
{
public:
    int languageCode;
    Programmer(int inpId)
    {
        id = inpId;
        languageCode = 9;
    }
    void getData(){
        cout<<id<<endl;
    }
};
```

Code Snippet 2: Inheritance Example Program

As shown in Code snippet 2,

- 1st we created an **"employee"** class which consists of public data member's integer **"id"** and float **"salary"**.
- 2nd the **"employee"** class consists of a parameterized constructor that takes an integer **"inpid"** parameter and assigns its value to the data member **"id"**. The value of variable **"salary"** is set to **"34"**.
- 3rd the **"employee"** class also consists of default constructor.
- 4th we created a **"programmer"** class that is inheriting **"employee"** class. The main thing to note here is that the **"visibility-mode"** is **"public"**.
- 5th the **"programmer"** class consists of public data member's integer **"languageCode"**.
- 6th the **"programmer"** class consists of a parameterized constructor that takes an integer **"inpid"** parameter and assigns its value to the data member **"id"**. The value of variable **"languageCode"** is set to **"9"**.

- 7th "programmer" class consists of a function **"getData"** which will print the value of the variable **"id"**.

  The main program is shown in code snippet 3.

```
int main()
{
Employee harry(1), rohan(2);
      cout << harry.salary << endl;
      cout << rohan.salary << endl;
Programmer skillF(10);
      cout << skillF.languageCode<<endl;
      cout << skillF.id<<endl;
skillF.getData();
return 0;
}
```

Code Snippet 3: Main Program

As shown in a code snippet 3,

- 1st objects **"harry"** and **"rohan"** is created of the **"employee"** data type. Object **"harry"** is passed with the value **"1"** and the object **"rohan"** is passed with the value **"2"**.
- 2nd the **"salary"** of both objects **"rohan"** and **"harry"** are printed.
- 3rd object **"skillF"** is created of the **"programmer"** data type. Object **"skillF"** is passed with the value **"10"**.
- 4th the **"languageCode"** and **"id"** of both object **"skillF"** is printed.
- 5th the function **"getData"** is called by the **"skillF"** object. This will print the **"id"**.

The output for the following program is shown in figure 1.

```
PS D:\MyData\Business\c
34
34
9
10
10
PS D:\MyData\Business\c
```

Figure 1: Program Output

Part: 38

Single Inheritance Deep Dive: Examples + Code

In this tutorial, we will discuss single inheritance in C++

Single Inheritance in C++

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class. For example, we have two classes **"employee"** and **"programmer"**. If the **"programmer"** class is inherited from the **"employee"** class which means that the **"programmer"** class can now implement the functionalities of the **"employee"** class.

An example program to demonstrate the concept of single inheritance in C++ is shown below.

```
class Base{
    int data1; // private by default and is not inheritable
public:
    int data2;
    void setData();
    int getData1();
    int getData2();
};
void Base ::setData(void){
    data1 = 10;
    data2 = 20;
}
int Base::getData1(){
    return data1;
}
int Base::getData2(){
    return data2;
}
```

Code Snippet 1: Base Class

As shown in a code snippet 1,

- 1st we created a **"base"** class which consists of private data member's integer **"data1"** and public data member integer **"data2"**.
- 2nd the **"base"** class consists of three members functions **"setData"**, **"getData1"**, and **"getData2"**.

- 3rd the function **"setData"** will assign the values **"10"** and **"20"** to the data members **"data1"** and **"data2"**.
- 4th the function **"getData1"** will return the value of the data member **"data1"**.
- 5th the function **"getData2"** will return the value of the data member **"data2"**.

The derived class will inherit the base class which is shown below.

```
class Derived: public Base{
// Class is being derived publically
    int data3;
public:
    void process();
    void display();
};
void Derived::process()
{
    data3 = data2 * getData1();
}
void Derived::display()
{
    cout << "Value of data 1 is " << getData1() << endl;
    cout << "Value of data 2 is " << data2 << endl;
    cout << "Value of data 3 is " << data3 << endl;
}
```

Code Snippet 2: Derived Class

As shown in Code snippet 2,

- 1st we created a **"derived"** class which is inheriting the base class publicly. The **"derived"** class consists of private data member's integer **"data3".**
- 2nd the **"derived"** class consists of two public member functions **"process"** and **"display"**.
- 3rd the function **"process"** will multiply the values **"data2"** and **"data1"**; and store the values in the variable **"data3"**.
- 4th the function **"display"** will print the values of the data member **"data1"**, **"data2"**, and **"data3"**.

The main program is shown in code snippet 3.

```
int main()
```

```
{
Derived der;
der.setData();
der.process();
der.display();
return 0;
}
```

Code Snippet 3: Main Program

As shown in a code snippet 3,

- 1st object **"der"** is created of the **"derived"** data type.
- 2nd the function **"setData"** is called by the object **"der"**. This function will set the values of the data members **"data1"** and **"data2"**
- 3rd the function **"process"** is called by the object **"der"**. This function will multiply the values **"data2"** and **"data1"**; and store their value in the variable **"data3"**.
- 4th the function **"display"** is called by the object **"der"**. This function will print the values of the data member **"data1"**, **"data2"**, and **"data3"**.

The output for the following program is shown in figure 1.

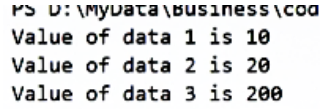

Figure 1: Program Output

Part: 39

Protected Access Modifier in C++

In this tutorial, we will discuss protected access modifiers in C++

Protected Access Modifiers in C++

Protected access modifiers are similar to the private access modifiers but protected access modifiers can be accessed in the derived class whereas private access modifiers cannot be accessed in the derived class. A table is shown below which shows the behaviour of access modifiers when they are derived **“`public`”**, **“`private`”**, and **“`protected`”**.

	Public Derivation	Private Derivation	Protected Derivation
Private members	Not Inherited	Not Inherited	Not Inherited
Protected members	Protected	Private	Protected
Public members	Public	Private	Protected

As shown in the table,

1. If the class is inherited in public mode then its private members cannot be inherited in child class.
2. If the class is inherited in public mode then its protected members are protected and can be accessed in child class.
3. If the class is inherited in public mode then its public members are public and can be accessed inside child class and outside the class.
4. If the class is inherited in private mode then its private members cannot be inherited in child class.
5. If the class is inherited in private mode then its protected members are private and cannot be accessed in child class.
6. If the class is inherited in private mode then its public members are private and cannot be accessed in child class.
7. If the class is inherited in protected mode then its private members cannot be inherited in child class.

8. If the class is inherited in protected mode then its protected members are protected and can be accessed in child class.
9. If the class is inherited in protected mode then its public members are protected and can be accessed in child class.

An example program to demonstrate the concept of protected access modifiers is shown below.

```
#include<iostream>
using namespace std;
class Base{
    protected:
        int a;
    private:
        int b;
};
class Derived: protected Base{
};
int main(){
    Base b;
    Derived d;
    // cout<<d.a; // Will not work since a is protected in both
base as well as derived class
    return 0;
}
```

Code Snippet 1: Protected Access Modifier Example Program

As shown in a code snippet 1,

- 1st we created a **"Base"** class which consists of protected data member integer **"a"** and private data member integer **"b"**.
- 2nd we created a **"Derived"** class which is inheriting the **"Base"** class in protected mode.
- 3rd the object **"b"** of the data type **"Base"** is created.
- 4th the object **"d"** of the data type **"Derived"** is created.
- 5th if we try to print the value of the data member **"a"** by using the object **"d"**; the program will throw an error because the data member **"a"** is protected and the derived class is inherited in the protected mode. So the data member **"a"** can only be accessed in the **"derived"** but not outside the class.

Part: 40

Multilevel Inheritance Deep Dive with Code Example in C++

In this tutorial, we will discuss multilevel inheritance in C++

Multilevel Inheritance in C++

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class. For example, we have three classes **"animal"**, **"mammal"** and **"cow"**. If the **"mammal"** class is inherited from the **"animal"** class and **"cow"** class is inherited from **"mammal"** which means that the **"mammal"** class can now implement the functionalities of **"animal"** and **"cow"** class can now implement the functionalities of **"mammal"** class.

An example program is shown below to demonstrate the concept of multilevel inheritance in C++.

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

class Student
{
protected:
    int roll_number;

public:
    void set_roll_number(int);
    void get_roll_number(void);
};

void Student ::set_roll_number(int r)
{
    roll_number = r;
}

void Student ::get_roll_number()
{
    cout << "The roll number is " << roll_number << endl;
}
```

Code Snippet 1: Student Class

As shown in a code snippet 1,

- 1st we created a **"student"** class which consists of protected data member integer **"roll_number"**.
- 2nd the **"student"** class consists of a public function **"set_roll_number"** and **"get_roll_number".**
- 3rd the function **"set_roll_number"** will set the value of the data member **"roll_number"**.
- 4th the function **"get_roll_number"** will print the value of the data member **"roll_number"**.

The code for the "exam" class is shown below which is inheriting the "student" class

```
class Exam : public Student
{
protected:
    float maths;
    float physics;
public:
    void set_marks(float, float);
    void get_marks(void);
};
void Exam ::set_marks(float m1, float m2)
{
    maths = m1;
    physics = m2;
}
void Exam ::get_marks()
{
    cout << "The marks obtained in maths are: " << maths << endl;
    cout << "The marks obtained in physics are: " << physics <<
endl;
}
```

Code Snippet 2: Exam Class

As shown in Code snippet 2,

- 1st we created an **"exam"** class that is inheriting **"student"** class in public mode.
- 2nd the **"exam"** class consists of protected data members float **"math"** and float **"physics"**.
- 3rd the **"exam"** class consists of public member functions **"set_marks"** and **"get_marks"**.

- 4th the function **"set_marks"** will set the value of the data members **"math"** and **"physics"**.
- 5th the function **"get_marks"** will print the value of the data members **"math"** and **"physics"**.

The code for the "result" class is shown below which is inheriting the "exam" class

```
class Result : public Exam{
    float percentage;
public:
    void display_results(){
        get_roll_number();
        get_marks();
        cout << "Your result is " << (maths + physics) / 2 << "%"
<< endl;
    }
};
```

Code Snippet 3: Result Class

As shown in a code snippet 3,

- 1st we created a **"Result"** class which is inheriting the **"Exam"** class in public mode.
- 2nd the **"Result"** class consists of private data member's float **"percentage"**.
- 3rd the **"exam"** class consists of the public member function **"display_results"**.
- 4th the function **"display_results"** will call the **"get_roll_number"** and **"get_marks"** functions, and add the values of **"math"** and **"physics"** variables then divide that value with **"2"** to get a percentage and prints it.

It can be clearly seen that the class **"Exam"** is inheriting class **"student"** and class **"Results"** is inheriting class **"Exam"**; which is an example of multilevel inheritance. The code main program is shown below.

```
int main()
{
    Result harry;
    harry.set_roll_number(420);
    harry.set_marks(94.0, 90.0);
    harry.display_results();
    return 0;
}
```

Code Snippet 4: Main Program

As shown in Code snippet 4,

- 1st object **"harry"** is created of the **"Result"** data type.
- 2nd the function **"set_roll_number"** is called by the object **"harry"** and the value **"420"** is passed.
- 3rd the function **"set_marks"** is called by the object **"harry"** and the values **"94.0"** and **"90.0"** are passed.
- 4th the function **"display_results"** is called by the object **"harry"**.

The output for the following program is shown in figure 1.

```
PS D:\MyData\Business\code playground\C
The roll number is 420
The marks obtained in maths are: 94
The marks obtained in physics are: 90
Your percentage is 92%
PS D:\MyData\Business\code playground\C
```

Figure 1: Program Output