INHERITANCE IN C++

What is Inheritance?

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Base Class: The class whose features are inherited is called the super class or the base class or the parent class.

Derived Class: The class that inherits the features is called the sub class or the derived class or the child class.

Advantages of Inheritance

1. code reusability and readability
2. The base class once defined and once it is compiled, it need not be reworked.
3. Saves time and effort as the main code need not be written again.

Defining Derived Class:

A derived class can be defined by specifying its relationship with the base class in addition to its own details.

Syntax

```
class derived_class_name: visibility mode base_class_name
{
        -----------
        -----------
        -----------
};
```

The colon (:) indicates that the derived_class_name is derived from the base_class_name. The visibility mode is optional and if present may be either public or private. The default visibility mode is private. Visibility mode specifies whether the feature of the base class are privately derived or publicly derived.

Access Control in C++

It is not possible for a derived class to directly access private members of its base class. However, it gets a complete object of base class which contains the private members. The derived class can also access private members of the base class through its public and protected members.

A derived class does not inherit

1. Constructors and destructors of the base class
2. Friend classes and friend functions of the base class

3. Overloaded operators of the base class

Modes of Inheritance

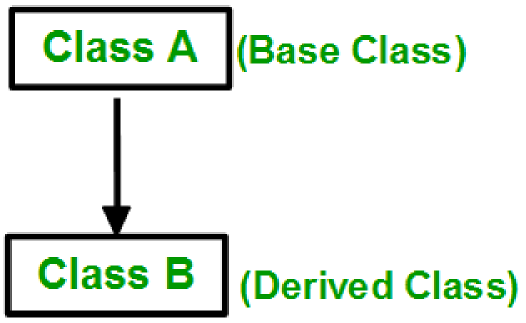
1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode:** If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode:** If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Multiple Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Single Inheritance: Public

```
#include<iostream>
using namespace std;
class base {
        private: int a;
        public: int b;
                void set_ab(int x, int y) {
                        a=x;
                        b=y;
                }
                int get_a() {
                        return a;
                }
                void show_a() {
                        cout<<"a="<<a<<endl;
                }
};
class derived: public base {
        private:
                int c;
        public:
                void mult() {
                        c=b*get_a();
                }
                void display() {
                        cout<<"a="<<get_a()<<endl;
                        cout<<"b="<<b<<endl;
                        cout<<"c="<<c<<endl;
                }
};
int main() {
        derived d1;
        d1.set_ab(10,20);
        d1.mult();
        d1.show_a();
        d1.display();

        d1.b=5;
        d1.mult();
        d1.display();
}
```

Single Inheritance: Private

```
#include<iostream>
using namespace std;
class base {
        private:
                int a;
        public:
                int b;
                void set_ab() {
                        cout<<"\n Enter Values a and b:";
                        cin>>a>>b;
                }
                int get_a() {
                        return a;
                }
                void show_a() {
                        cout<<"a="<<a<<endl;
                }
};
class derived: private base {
        private:
                int c;
        public:
                void mult() {
                        set_ab();
                        c=b*get_a();
                }
                void display() {
                        show_a();
                        cout<<"b="<<b<<endl;
                        cout<<"c="<<c<<endl;
                }
};
int main() {
        derived d1;
        //d1.set_ab(); //Error can not access private member 'a'.
        d1.mult();
        //d1.show_a();
        d1.display();

        //d1.b=5;
        d1.mult();
        d1.display();
}
```

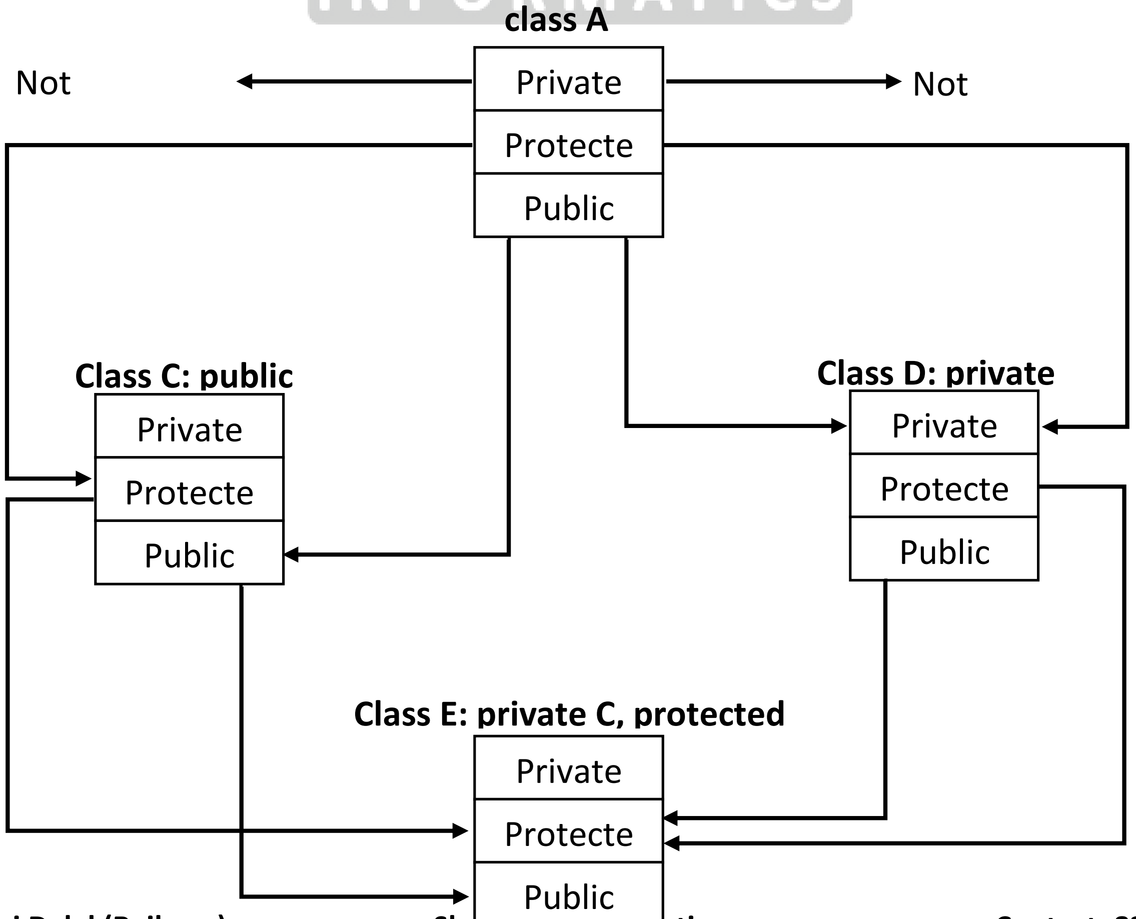
Making a private member inheritable:

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What we do if the private data needs to the private member by making it public. This would make it accessible to all the other function of the program, thus taking away the advantage of data hiding.

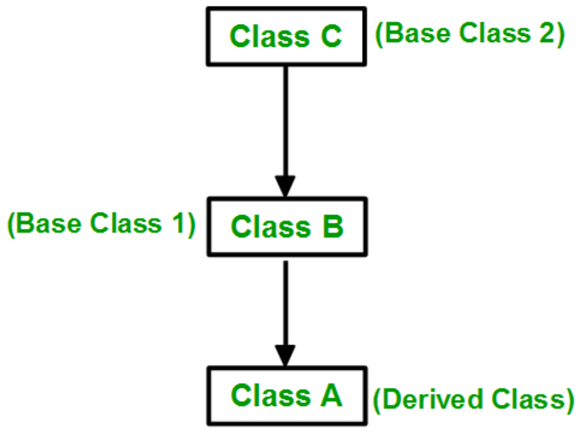
C++ provides a third visibility modifier protected which serve a limited purpose in inheritance. A member declared as protected is accessible by the member function within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as

```
class alpha
{
        private:                        //optional
                -------                 //only visible to member function within its class
                -------
        Protected:
                -------                 //visible to member functions of its own class and derived class
                -------
        Public:
                -------                 //visible to all functions in the program
                -------
}
```

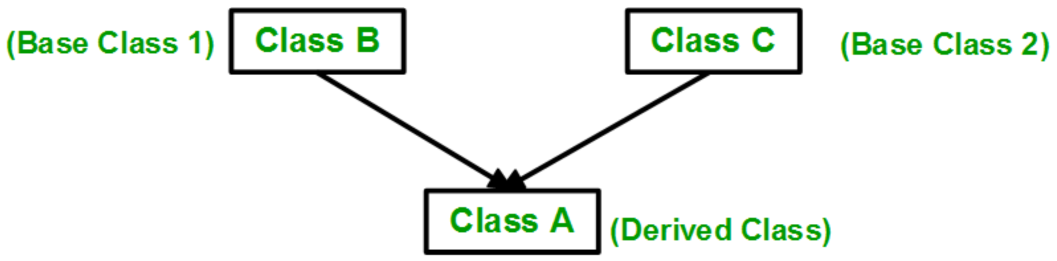
When a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since private member cannot be inherited).



2. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



3. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Example Program

```
#include<iostream>
using namespace std;
class base1
{
        protected:
                int x;
        public:
                void get_x(int a)
                {
                        x=a;
                }
};
class base2
{
        protected:
                int y;
        public:
                void get_y(int b)
                {
                        y=b;
                }
};
class derived: public base1, public base2
{
        protected:
                int z;
        public:
                void display()
                {
                        z=x+y;
                        cout<<endl<<"X="<<x;
                        cout<<endl<<"Y="<<y;
                        cout<<endl<<"Z="<<z;
                }
};
int main()
{
        derived d1;
        d1.get_x(5);
        d1.get_y(10);
        d1.display();
}
```

Ambiguity Resolution in Inheritance

Sometime, we may face a problem in using multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes as

```
class A
{
        public:
            void display()
            {
                cout<<"class A";
            }
};
class B: public A
{
        public:
            void display()
            {
                cout<<"class B";
            }
};
```

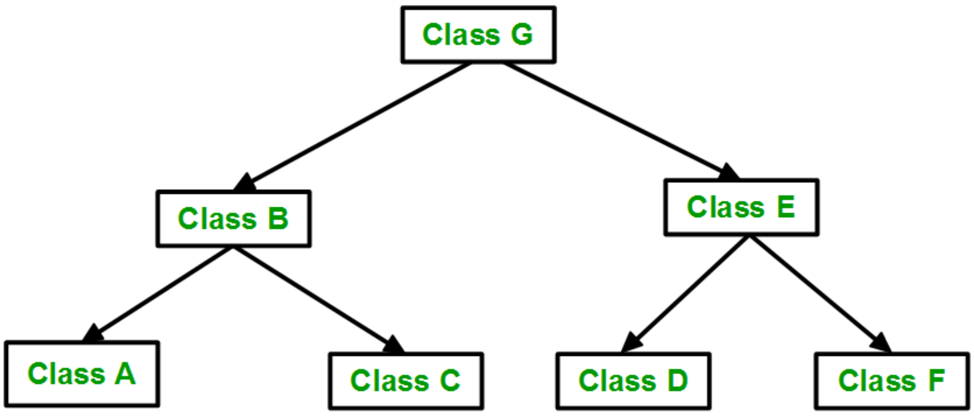
Which display() function used by the derived class when we inherit these two classes?

We can solve this problem by defining a named instance within the derived class, using the class scope resolution operator(::) with the function as

```
#include<iostream>
using namespace std;
class A
{
        public:
            void display()
            {
                cout<<endl<<"class A";
```

```
                    }
};
class B: public A
{
          public:
                    void display()
                    {
                              cout<<endl<<"class B";
                    }
};
int main()
{
          B b;
          b.display();
          b.A::display();
          b.B::display();
}
```

4. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

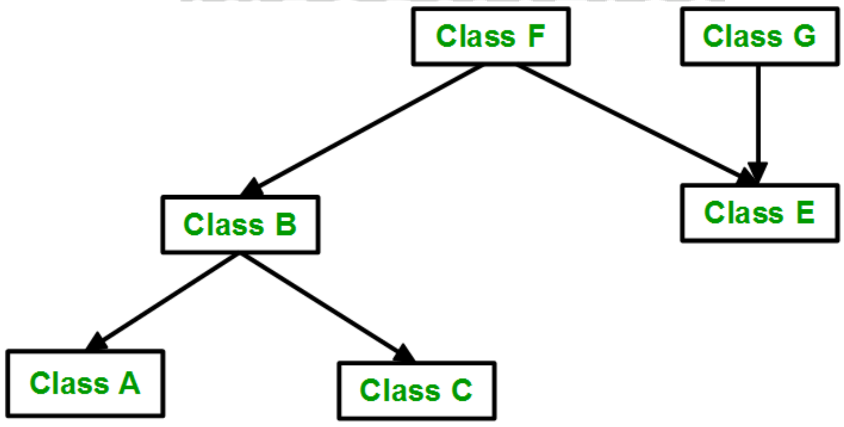


Example Program:

```
#include <iostream>
using namespace std;
class A
{
   public:
                              int x, y;
                              void getdata()
                              {
                                 cout<<endl<<"Enter value of x and y:";
                                 cin>>x>>y;
                              }
};
class B : public A
{
   public:
                              void product()
                              {
                                cout<<endl<<"Product= "<<x*y;
                            }
};
class C : public A
{
   public:
                            void sum()
                            {
                        cout <<endl<<"Sum="<<x+y;
                            }
};
int main()
{
   B obj1;
   C obj2;
   obj1.getdata();
   obj1.product();
   obj2.getdata();
   obj2.sum();

}
```

5. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



Example Program:

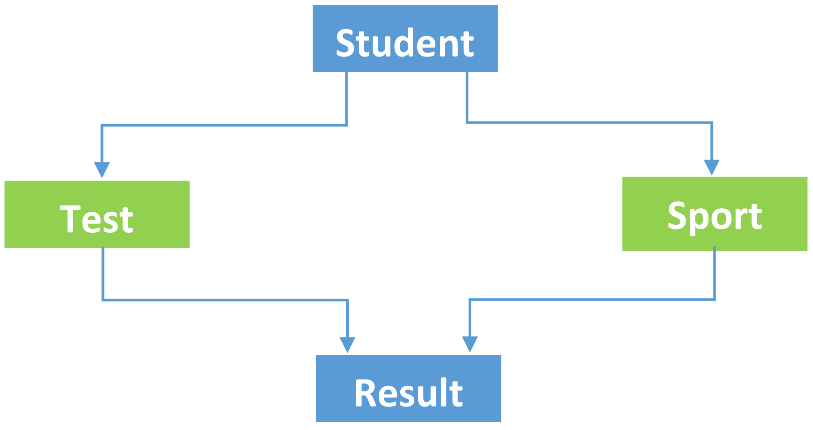
```
#include<iostream>
using namespace std;
class Student
{
          protected:
                              int rno;
                              string sname;
          public:
                              void getData()
                              {
```

```
				cout<<endl<<"Enter Roll No:";
				cin>>rno;
				cout<<endl<<"Enter Student Name:";
				cin>>sname;
			}
			void showData()
			{
				cout<<endl<<"Roll No:"<<rno;
				cout<<endl<<"Student Name:"<<sname;
			}
};
class Test: public Student
{
	protected:
			int m1,m2,m3;
	public:
			void getMarks()
			{
				cout<<endl<<"Enter marks of 3 subject:";
				cin>>m1>>m2>>m3;
			}
			void showMarks()
			{
				cout<<endl<<"M-I:"<<m1;
				cout<<endl<<"M-II:"<<m2;
				cout<<endl<<"M-III:"<<m3;
			}
};
class Sport
{
	protected:
			char grade;
	public:
			void getGrade()
			{
				cout<<endl<<"Enter Grade:";
				cin>>grade;
			}
			void showGrade()
			{
				cout<<endl<<"Grade:"<<grade;
			}
};
class Result: public Test, public Sport
{
	private:
```

```
                    int total;
                    float ptage;
          public:
                    void displayResult()
                    {
                              showData();
                              showMarks();
                              showGrade();
                              total=m1+m2+m3;
                              ptage=(float)total/3;
                              cout<<endl<<"Total:"<<total;
                              cout<<endl<<"Percentage:"<<ptage;
                    }
};
int main()
{
          Result r1;
          r1.getData();
          r1.getMarks();
          r1.getGrade();
          r1.displayResult();
}
```

Virtual Base Class:

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class. C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.



Example program:

```
#include<iostream>
using namespace std;
class Student
{
          protected:
                              int rno;
                              string sname;
```

```
        public:
                        void getData()
                        {
                                cout<<endl<<"Enter Roll No:";
                                cin>>rno;
                                cout<<endl<<"Enter Student Name:";
                                cin>>sname;
                        }
                        void showData()
                        {
                                cout<<endl<<"Roll No:"<<rno;
                                cout<<endl<<"Student Name:"<<sname;
                        }
};
class Test: virtual public Student
{
        protected:
                        int m1, m2, m3;
        public:
                        void getMarks()
                        {
                                cout<<endl<<"Enter marks of 3 subject:";
                                cin>>m1>>m2>>m3;
                        }
                        void showMarks()
                        {
                                cout<<endl<<"M-I:"<<m1;
                                cout<<endl<<"M-II:"<<m2;
                                cout<<endl<<"M-III:"<<m3;
                        }
};
class Sport: virtual public Student
{
        protected:
                        char grade;
        public:
                        void getGrade()
                        {
                                cout<<"\n Enter Grade:";
                                cin>>grade;
                        }
                        void showGrade()
                        {
                                cout<<endl<<"Grade:"<<grade;
                        }
};
```

```
class Result: public Test, public Sport
{
        private:
                        int total;
                        float ptage;
        public:
                        void displayResult()
                        {
                                showData();
                                showMarks();
                                showGrade();
                                total=m1+m2+m3;
                                ptage=(float)total/3.0;
                                cout<<endl<<"Total:"<<total;
                                cout<<endl<<"Percentage:"<<ptage;
                        }
};

int main()
{
        Result r1;
        r1.getData();
        r1.getMarks();
        r1.getGrade();
        r1.displayResult();
}
```

Constructor in Derived Class:

A constructor plays a vital role in initializing an object. An important note, while using constructors during inheritance, is that, as long as a base class constructor does not take any arguments, the derived class need not have a constructor function. However, if a base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor. Remember, while applying inheritance, we usually create objects using derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base class contains constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base class is constructed in the same order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructor will be executed in the order of inheritance.

The derived class takes the responsibility of supplying the initial values to its base class. The constructor of the derived class receives the entire list of required values as its argument and passes them on to the base constructor in the order in which they are declared in the derived class. A base class constructor is called and executed before executing the statements in the body of the derived class.

The header line of the derived-constructor function contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the derived class constructor and the second part lists the function calls to the base class.

Example Program:

```
// program to show how constructors are invoked in derived class
#include <iostream>
using namespace std;
class alpha
{
   private:
                  int x;
   public:
               alpha(int i)
               {
                  x = i;
                  cout<<endl<<"alpha initialized";
               }
               void show_x()
               {
                  cout<<endl<<"x="<<x;
               }
};
class beta
{
   private:
                  float y;
   public:
               beta(float j)
               {
                  y = j;
                  cout<<endl<<"beta initialized";
               }
               void show_y()
               {
                  cout<<endl<<"y = "<<y;
               }
};
class gamma : public beta, public alpha
{
   private:
                  int n,m;
   public:
               gamma(int a, float b, int c, int d):alpha(a), beta(b)
               {
                  m = c;
                  n = d;
```

```
                cout<<endl<<"gamma initialized";
            }
            void show_mn()
            {
                cout<<endl<<"m = "<<m;
                cout<<endl<<"n = "<<n;
            }
};

int main()
{
    gamma g(5, 7.65, 30, 100);
    g.show_x();
    g.show_y();
    g.show_mn();
}
```

Friend Function:

Friend functions of the class are granted permission to access private and protected members of the class in C++. They are defined globally outside the class' scope. Friend functions are not member functions of the class. So, what exactly is the friend function?

A friend function is a function that is declared outside a class, but is capable of accessing the private and protected members of class. There could be situations in programming wherein we want two classes to share their members. These members may be data members, class functions or function templates. In such cases, we make the desired function, a friend to both these classes which will allow to access private and protected data members of the class.

Generally, non-member functions cannot access the private members of a particular class. Once declared as a friend function, the function is able to access the private and the protected members of these classes.

Syntax

```
class className {
    ... .. ...
    friend returntype functionName(arguments);
    ... .. ...
}
```

Characteristics of Friend Function in C++

1. The function is not in the 'scope' of the class to which it has been declared a friend.
2. Friend functionality is not restricted to only one class
3. Friend functions can be a member of a class or a function that is declared outside the scope of class.
4. It cannot be invoked using the object as it is not in the scope of that class.
5. We can invoke it like any normal function of the class.
6. Friend functions have objects as arguments.

7. It cannot access the member names directly and has to use dot membership operator and use an object name with the member name.
8. We can declare it either in the ‘public’ or the ‘private’ part.

Example Program

```
#include<iostream>
using namespace std;
class Sample
{
        private:
                        int a, b;
        public:
                        void setValue()
                        {
                                a=2;
                                b=5;
                        }
                        friend float mean(Sample s);
};
float mean(Sample s)
{
        return float(s.a+s.b)/2.0;
}
int main()
{
        Sample x;
        x.setValue();
        cout<<endl<<"Mean Value:"<<mean(x);
}
```

Friend Function with Two Classes:

```
#include<iostream>
using namespace std;
class ABC;
class XYZ
{
        private:
                        int x;
        public:
                        void setValue(int i)
                        {
                                x=i;
                        }
                        friend void max(XYZ, ABC);
};
class ABC
```

```
{
	private:
			int a;
	public:
			void setValue(int i)
			{
				a=i;
			}
			friend void max(XYZ, ABC);
};
void max(XYZ m, ABC n)
{
	if(m.x>=n.a)
		cout<<endl<<"Maximum:"<<m.x;
	else
		cout<<endl<<"Maximum:"<<n.a;
}
int main()
{
	ABC abc;
	abc.setValue(7);
	XYZ xyz;
	xyz.setValue(13);
	max(xyz, abc);
}
```