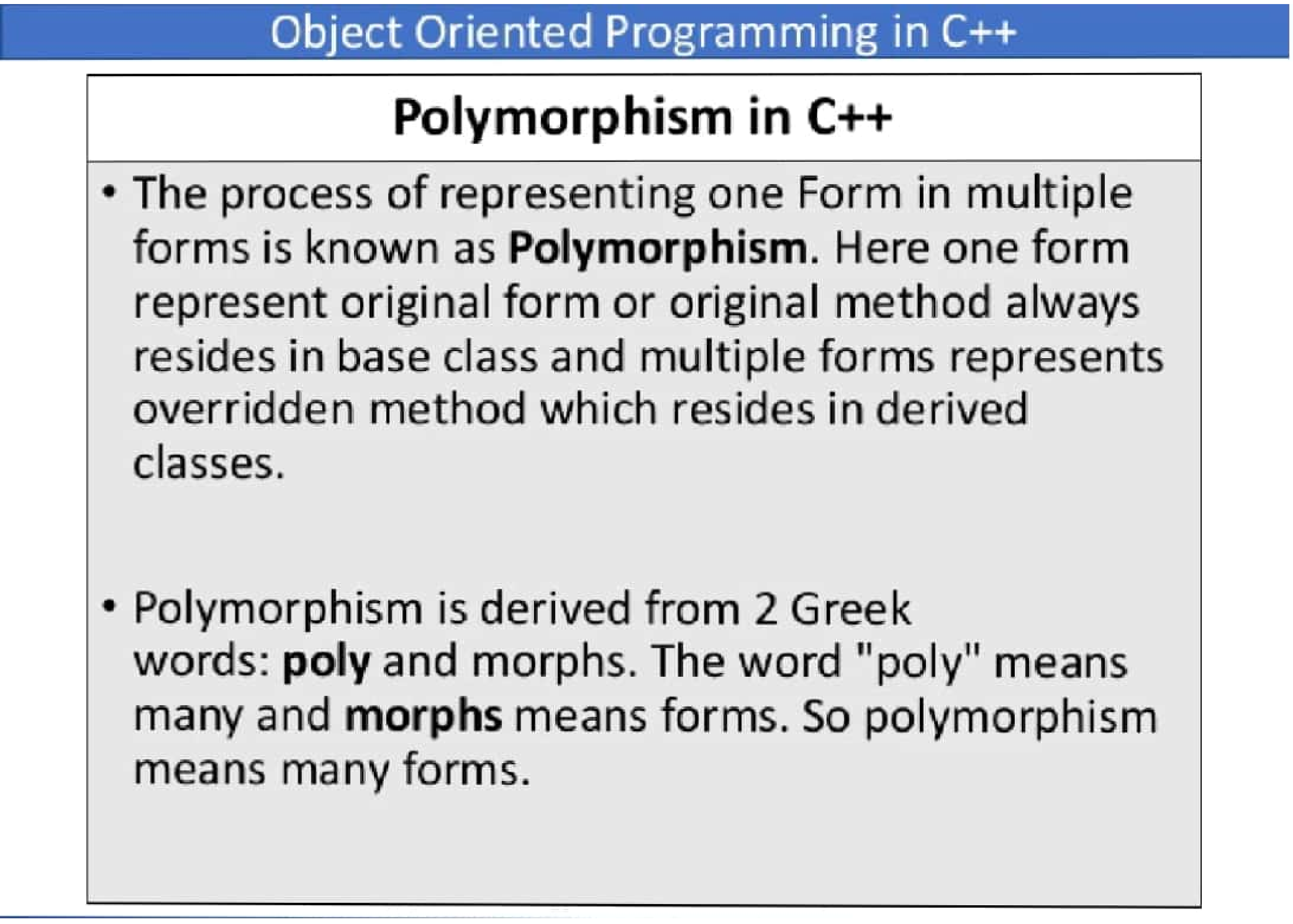
Polymorphism in C++



Polymorphism in C++

- The process of representing one Form in multiple forms is known as **Polymorphism**. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

- Polymorphism is derived from 2 Greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.

Polymorphism in C++

Real life example of Polymorphism in C++

- Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

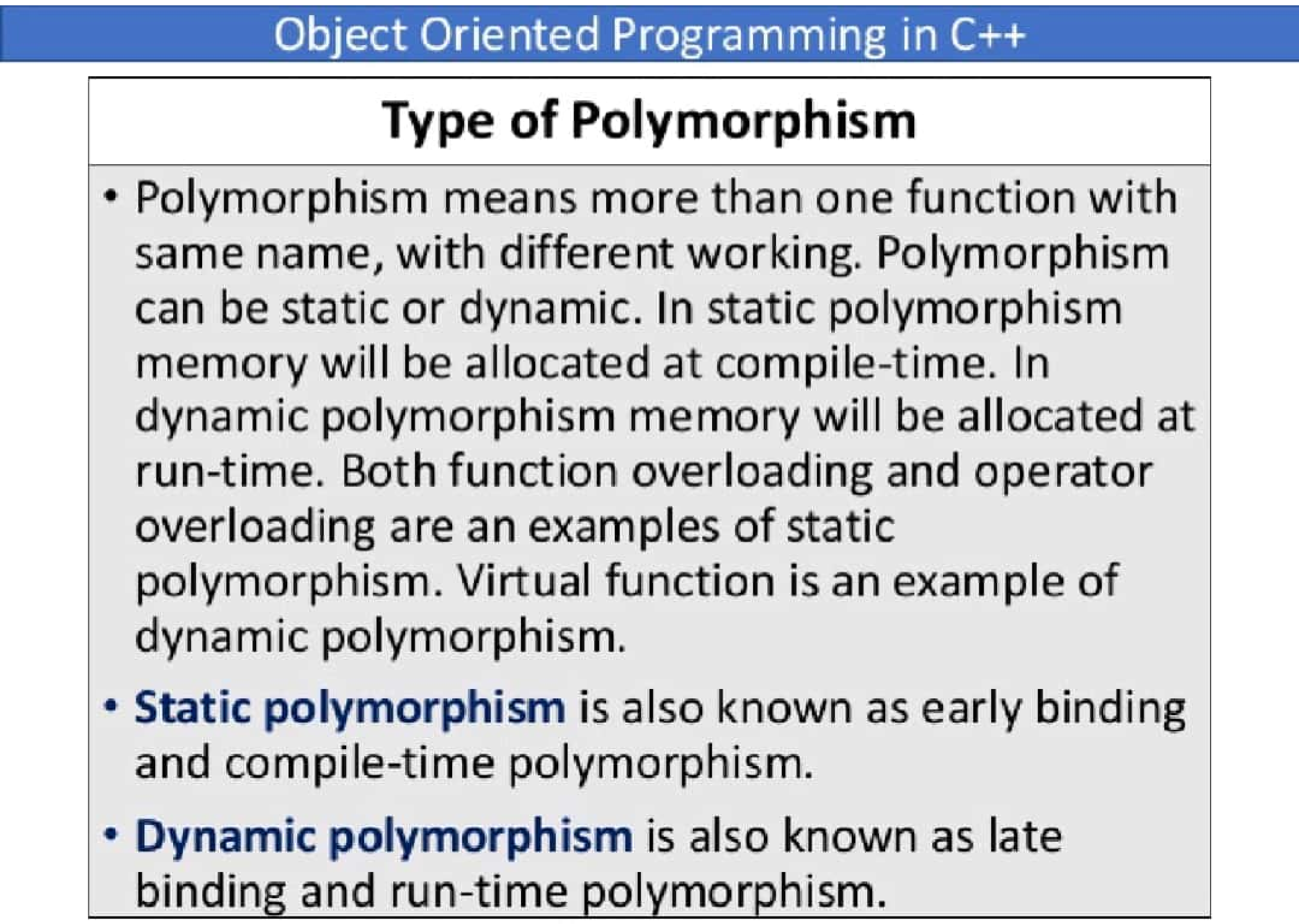
Polymorphism in C++

In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son



Type of Polymorphism

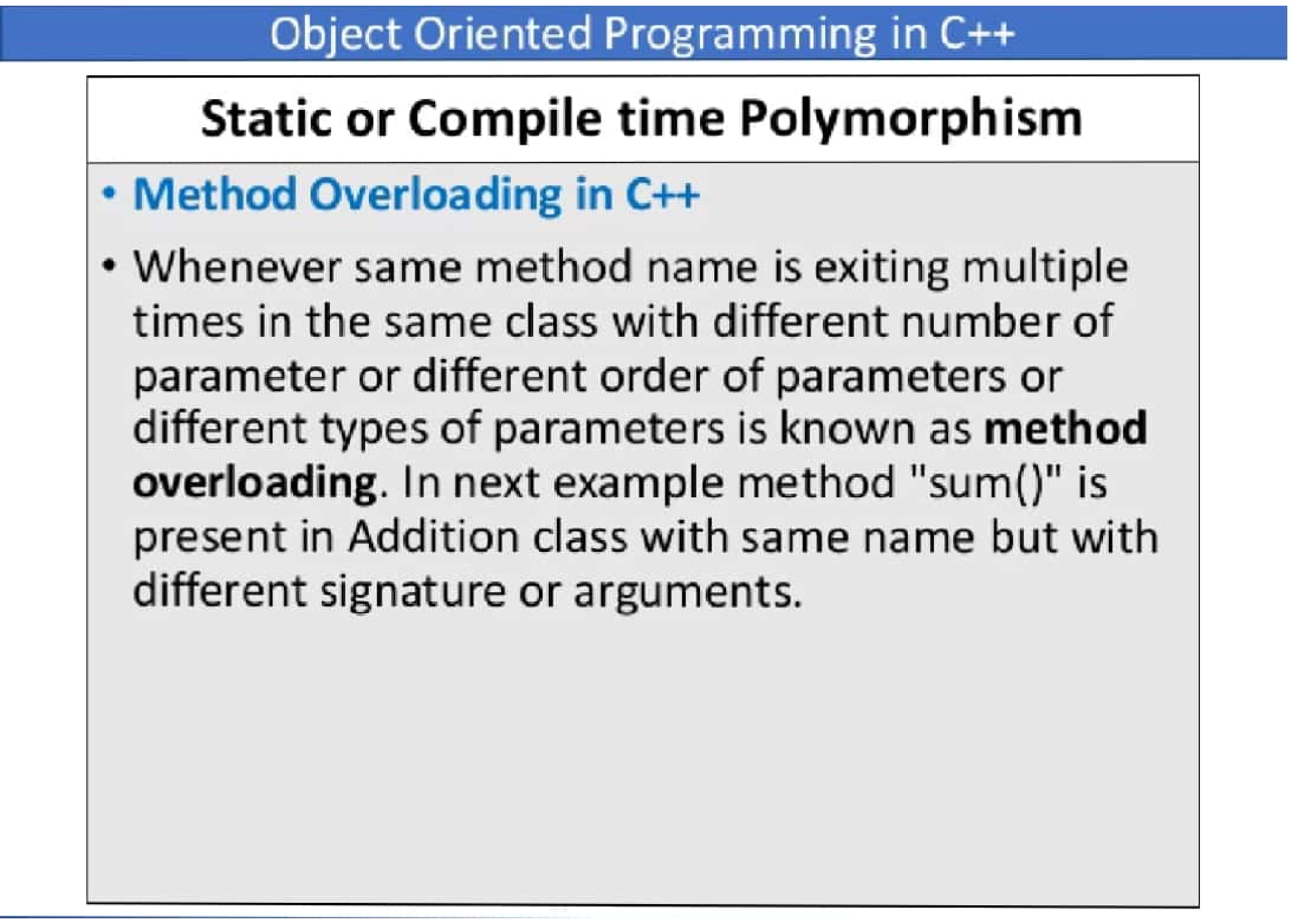
- Polymorphism means more than one function with same name, with different working. Polymorphism can be static or dynamic. In static polymorphism memory will be allocated at compile-time. In dynamic polymorphism memory will be allocated at run-time. Both function overloading and operator overloading are an examples of static polymorphism. Virtual function is an example of dynamic polymorphism.
- **Static polymorphism** is also known as early binding and compile-time polymorphism.
- **Dynamic polymorphism** is also known as late binding and run-time polymorphism.

Type of Polymorphism

- **Compile time polymorphism:**
  - In this method object is bound to the function call at the compile time itself.
- **Run time polymorphism:**
  - In this method object is bound to the function call only at the run time.

Type of Polymorphism

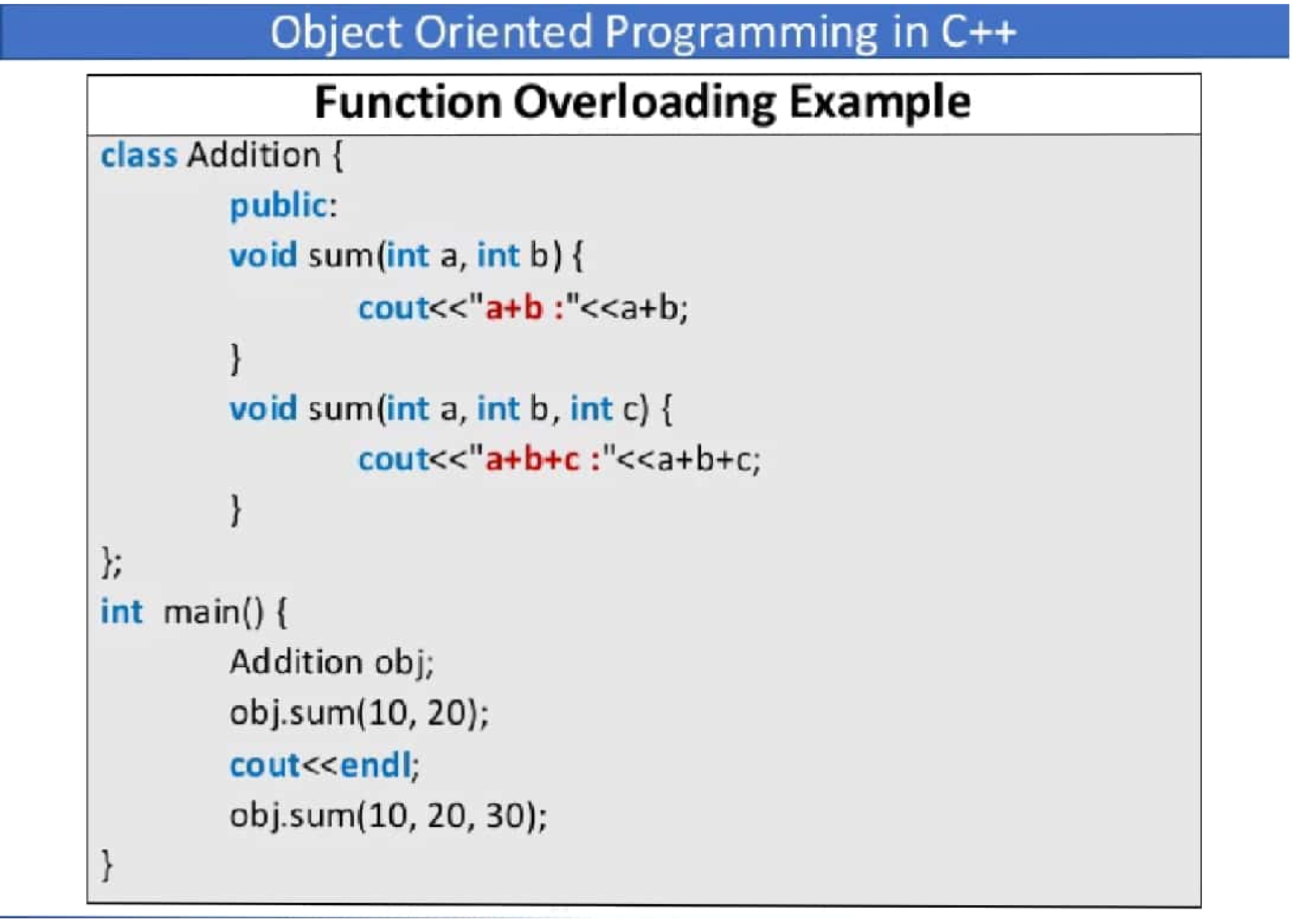
- **Compile time polymorphism**
- In C++ programming you can achieve compile time polymorphism in two way, which is given below;
  - Method Overloading
  - Method Overriding



Object Oriented Programming in C++

Static or Compile time Polymorphism

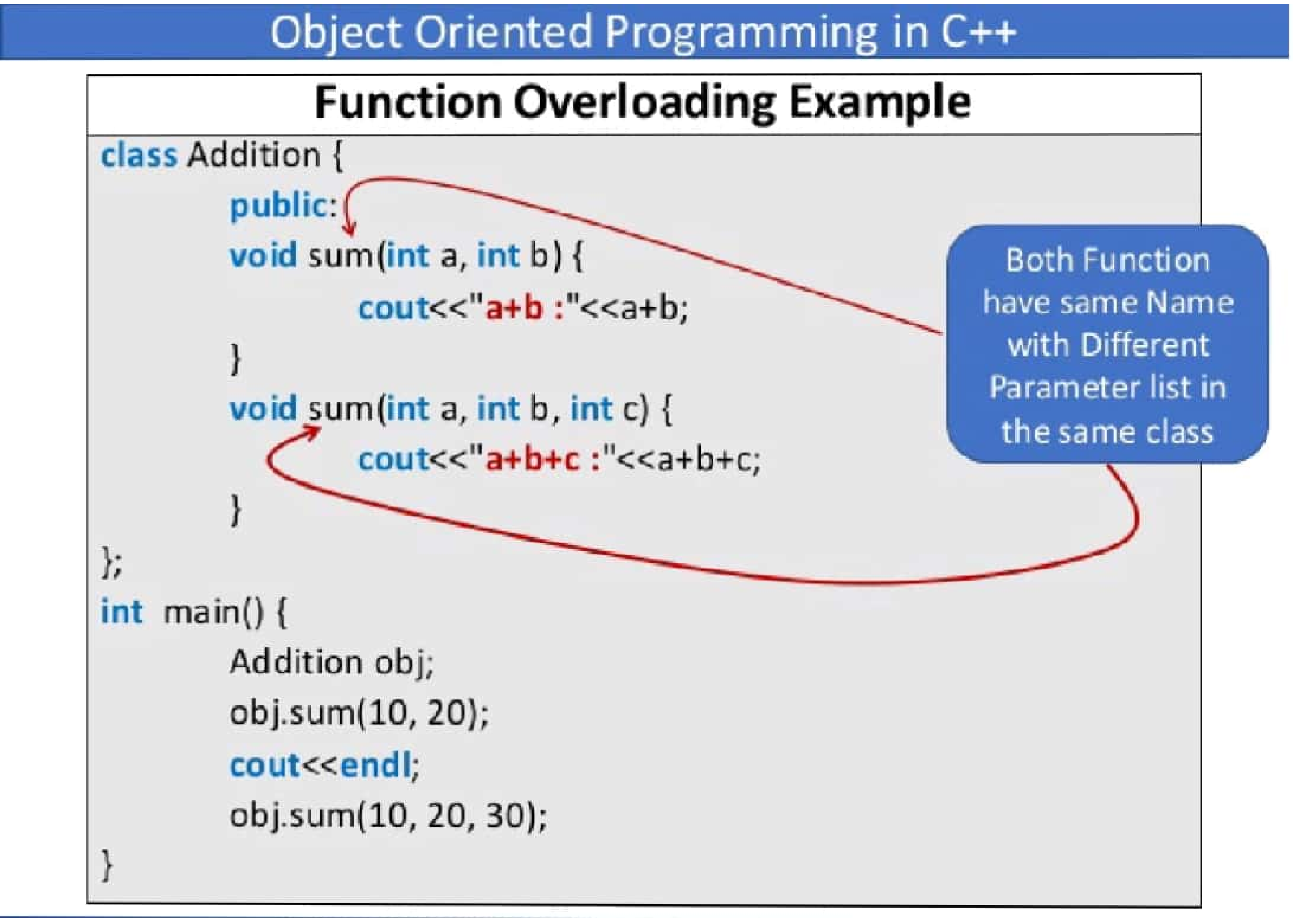
- Method Overloading in C++
- Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**. In next example method "sum()" is present in Addition class with same name but with different signature or arguments.



Object Oriented Programming in C++

Function Overloading Example

```
class Addition {
          public:
          void sum(int a, int b) {
                    cout<<"a+b :"<<a+b;
          }
          void sum(int a, int b, int c) {
                    cout<<"a+b+c :"<<a+b+c;
          }
};
int  main() {
          Addition obj;
          obj.sum(10, 20);
          cout<<endl;
          obj.sum(10, 20, 30);
}
```

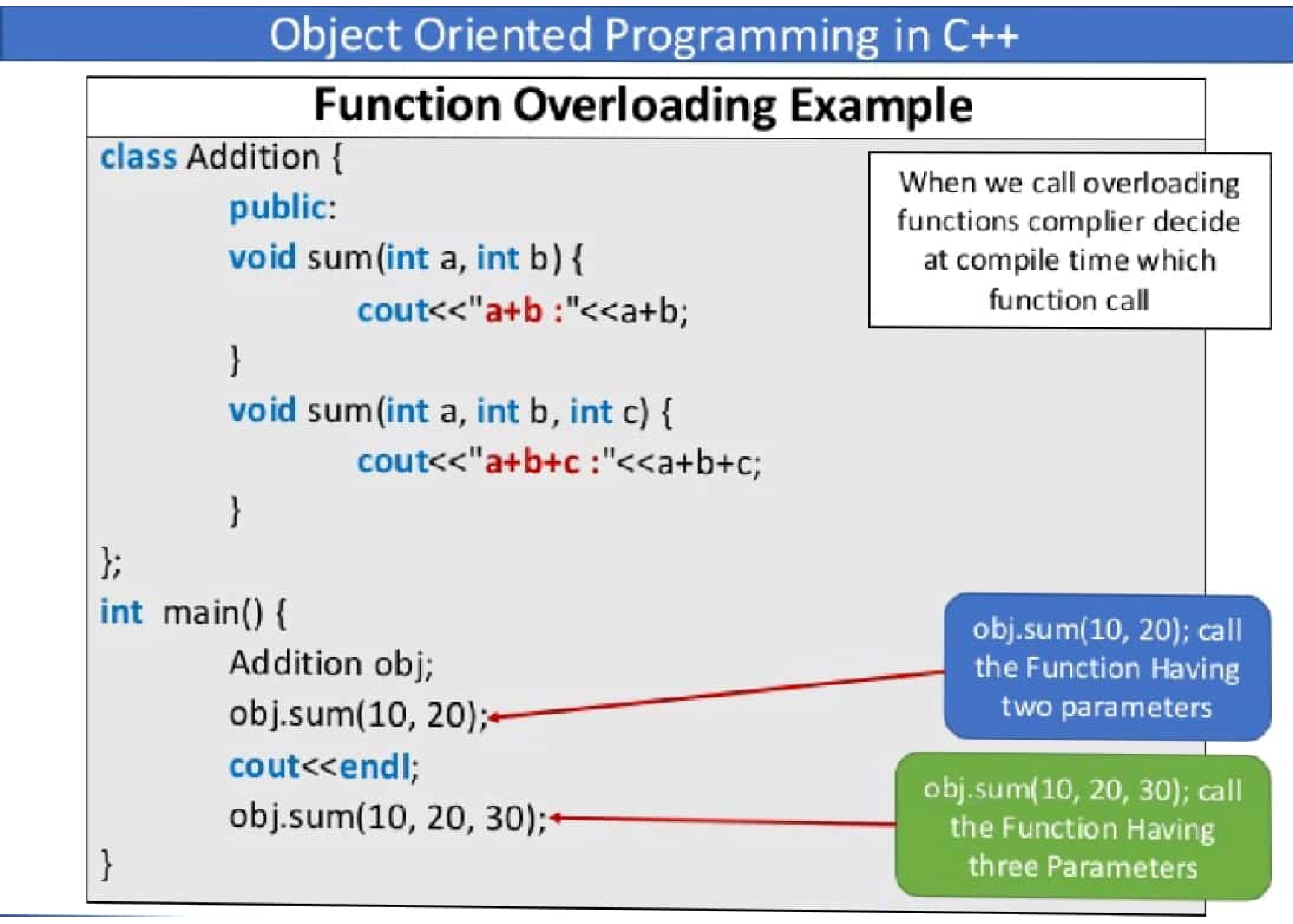


Object Oriented Programming in C++

Function Overloading Example

```
class Addition {
          public:
          void sum(int a, int b) {
                    cout<<"a+b :"<<a+b;
          }
          void sum(int a, int b, int c) {
                    cout<<"a+b+c :"<<a+b+c;
          }
};
int  main() {
          Addition obj;
          obj.sum(10, 20);
          cout<<endl;
          obj.sum(10, 20, 30);
}
```

Both Function have same Name with Different Parameter list in the same class



Object Oriented Programming in C++

Function Overloading Example

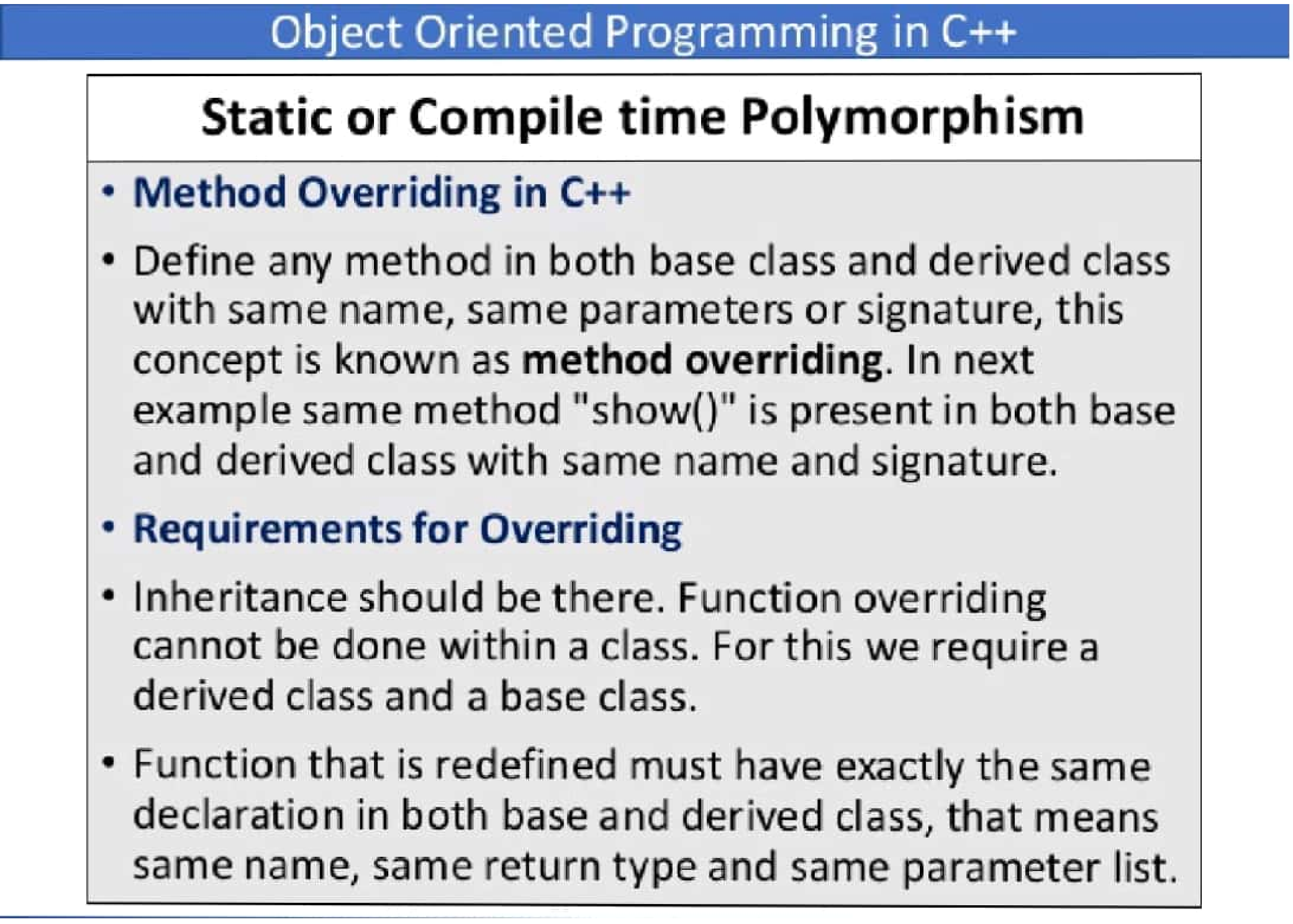
```
class Addition {
        public:
        void sum(int a, int b) {
                cout<<"a+b :"<<a+b;
        }
        void sum(int a, int b, int c) {
                cout<<"a+b+c :"<<a+b+c;
        }
};
int  main() {
        Addition obj;
        obj.sum(10, 20);
        cout<<endl;
        obj.sum(10, 20, 30);
}
```

When we call overloading functions complier decide at compile time which function call

obj.sum(10, 20); call the Function Having two parameters

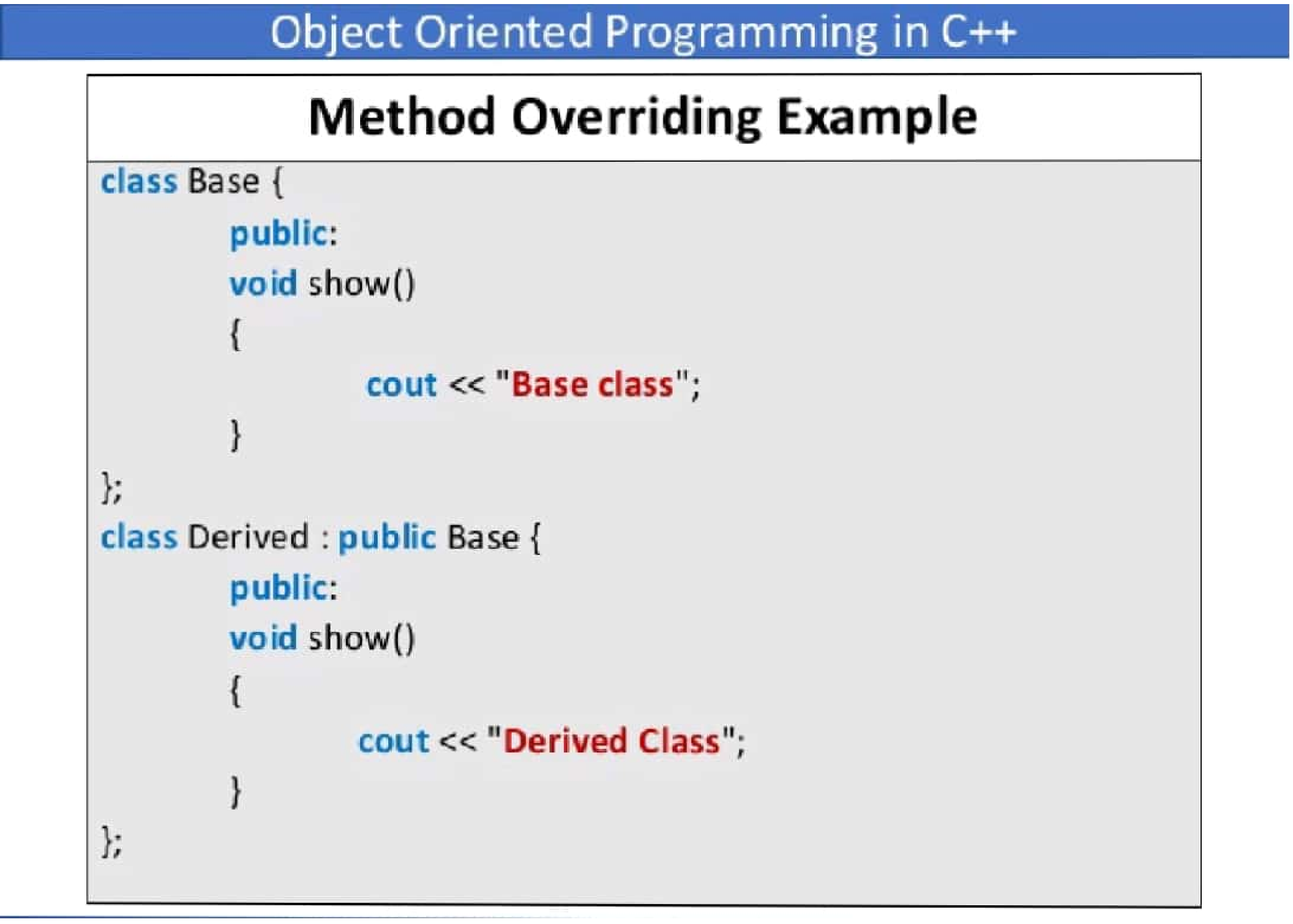
obj.sum(10, 20, 30); call the Function Having three Parameters

Scanned by CamScanner



Static or Compile time Polymorphism

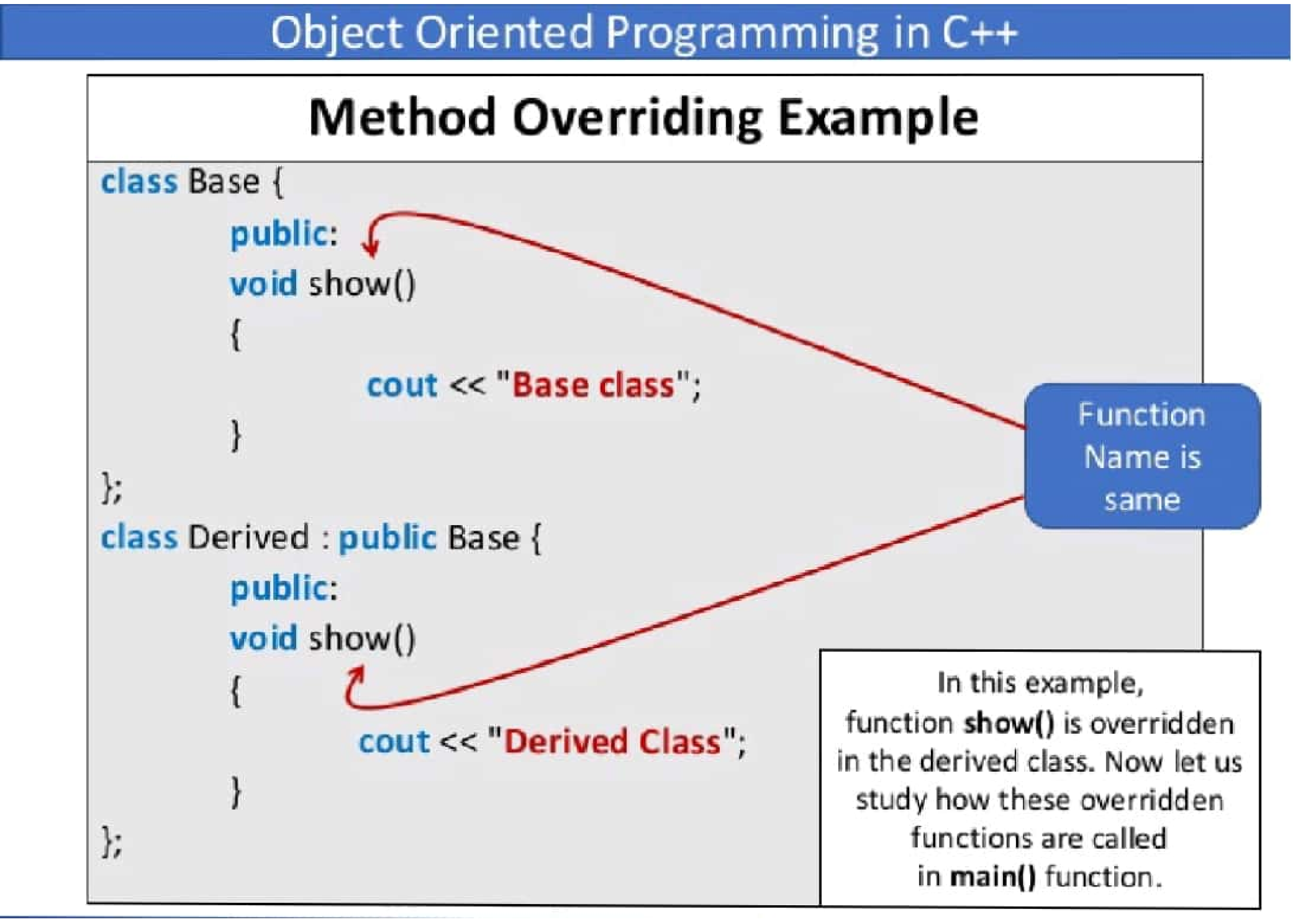
- **Method Overriding in C++**
- Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**. In next example same method "show()" is present in both base and derived class with same name and signature.
- **Requirements for Overriding**
- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.



Object Oriented Programming in C++

Method Overriding Example

```
class Base {
        public:
        void show()
        {
                cout << "Base class";
        }
};
class Derived : public Base {
        public:
        void show()
        {
               cout << "Derived Class";
        }
};
```



Object Oriented Programming in C++

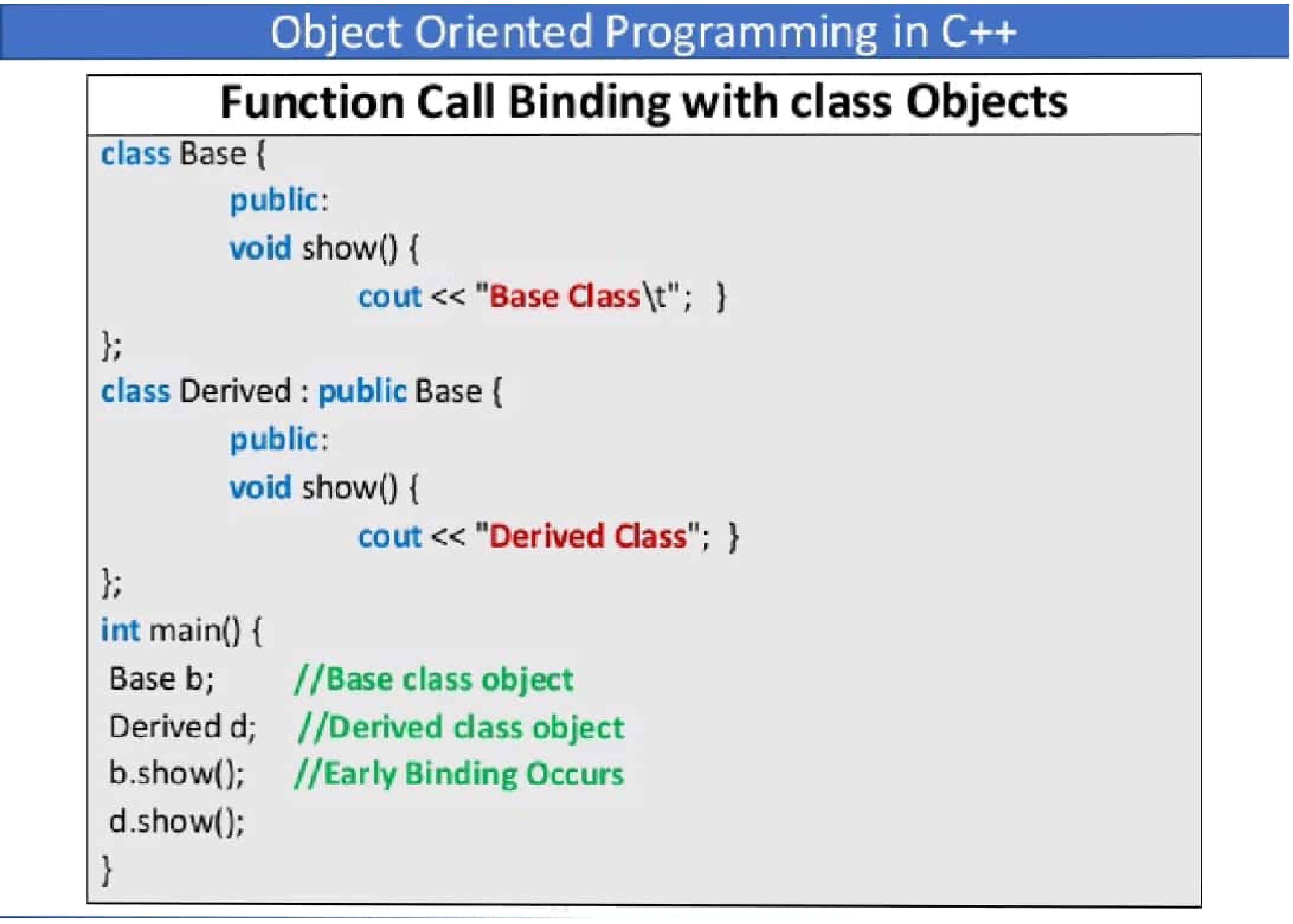
Method Overriding Example

```
class Base {
        public:
        void show()
        {
                cout << "Base class";
        }
};
class Derived : public Base {
        public:
        void show()
        {
                cout << "Derived Class";
        }
};
```

In this example, function **show()** is overridden in the derived class. Now let us study how these overridden functions are called in **main()** function.

Function Call Binding with class Objects

- Connecting the function call to the function body is called **Binding**. When it is done before the program is run, its called **Early** Binding or **Static** Binding or **Compile-time** Binding.

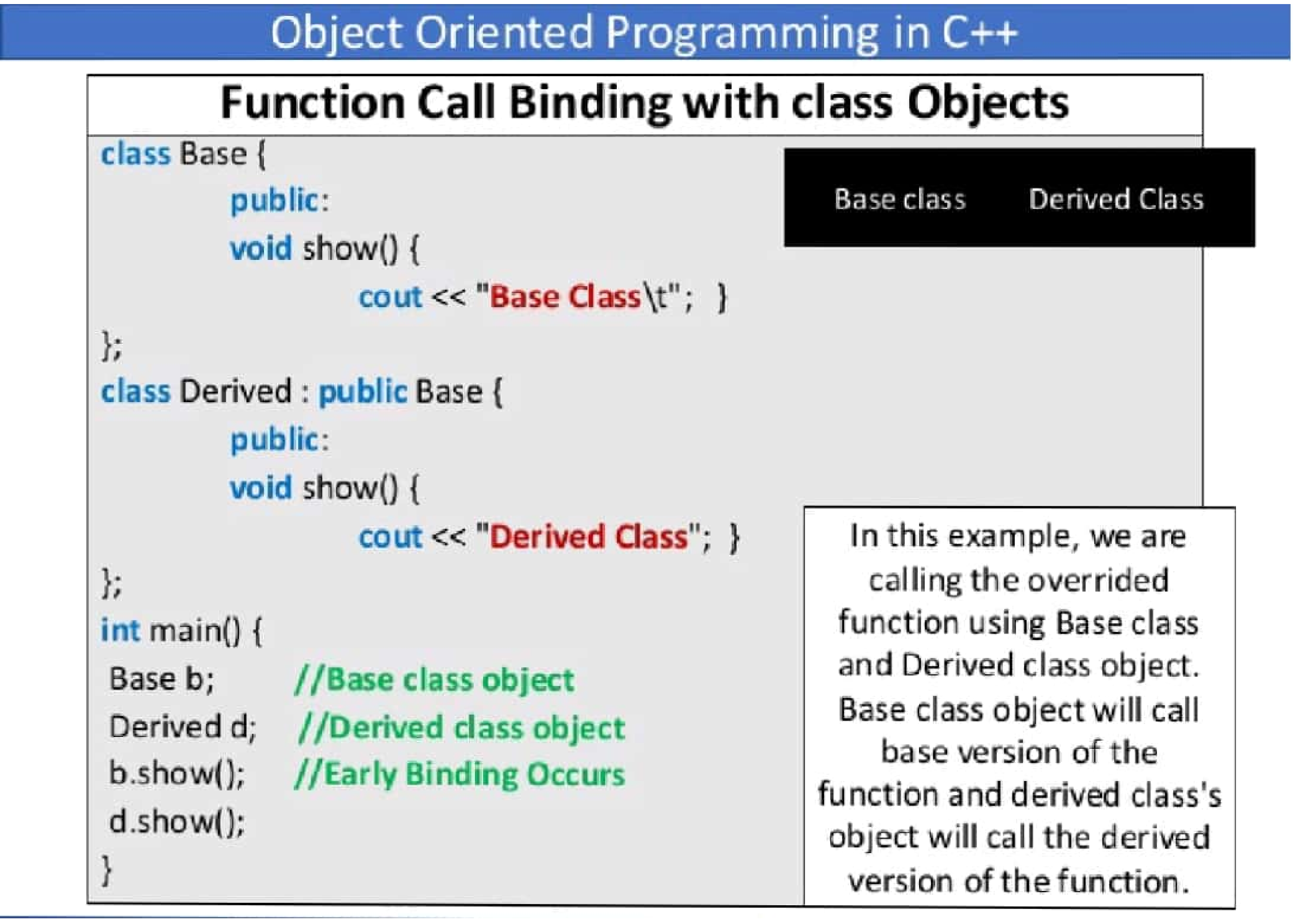


Object Oriented Programming in C++

Function Call Binding with class Objects

```
class Base {
            public:
            void show() {
                        cout << "Base Class\t";   }
};
class Derived : public Base {
            public:
            void show() {
                        cout << "Derived Class";  }
};
int main() {
 Base b;          //Base class object
 Derived d;       //Derived class object
 b.show();        //Early Binding Occurs
 d.show();
}
```

Scanned by CamScanner



Object Oriented Programming in C++

Function Call Binding with class Objects

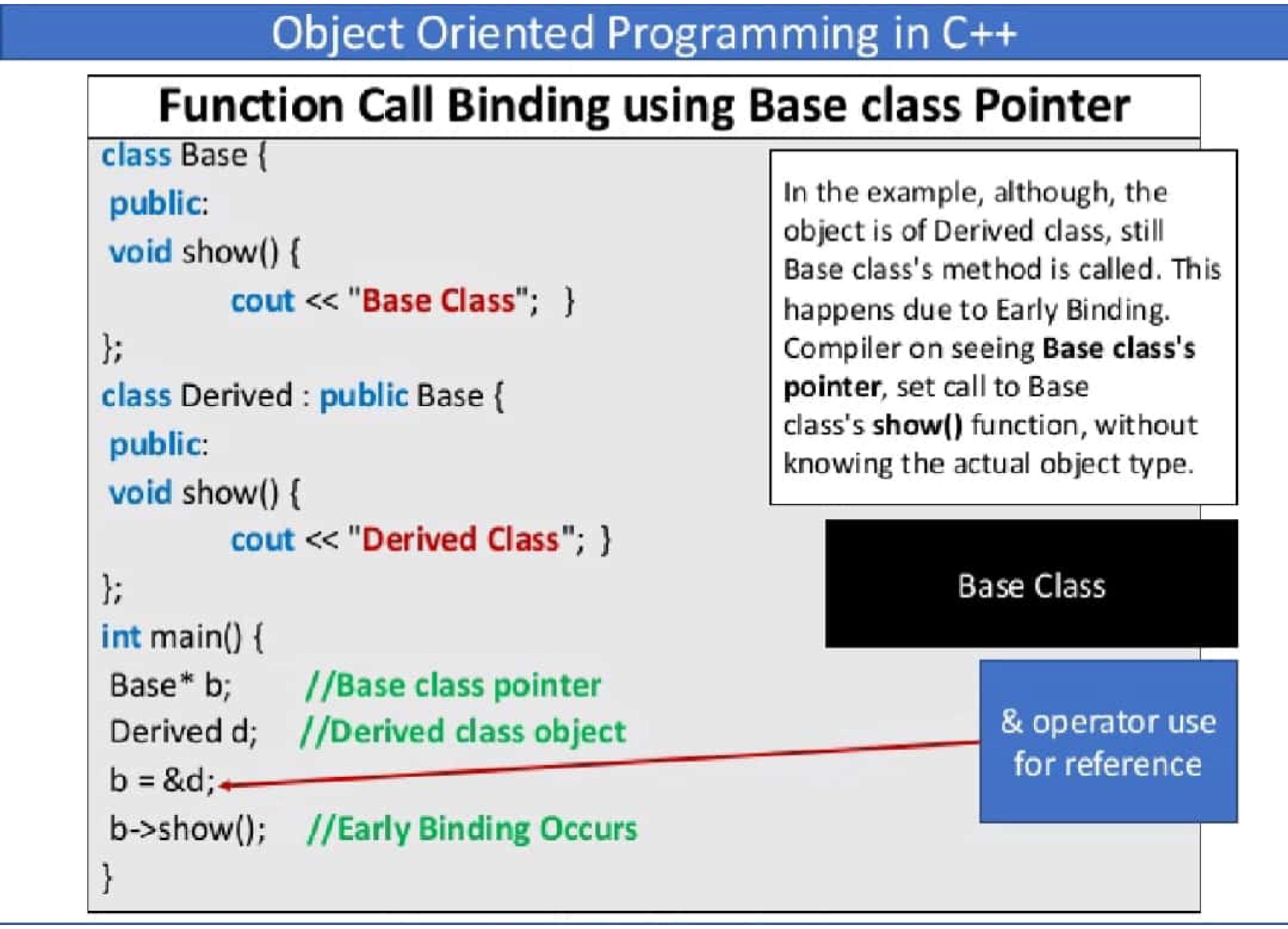
```
class Base {
        public:
        void show() {
                cout << "Base Class\t";  }
};
class Derived : public Base {
        public:
        void show() {
                cout << "Derived Class";  }
};
int main() {
 Base b;        //Base class object
 Derived d;     //Derived class object
 b.show();      //Early Binding Occurs
 d.show();
}
```

```
Base class      Derived Class
```

In this example, we are calling the overrided function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

Function Call Binding using Base class Pointer

- But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.



Object Oriented Programming in C++

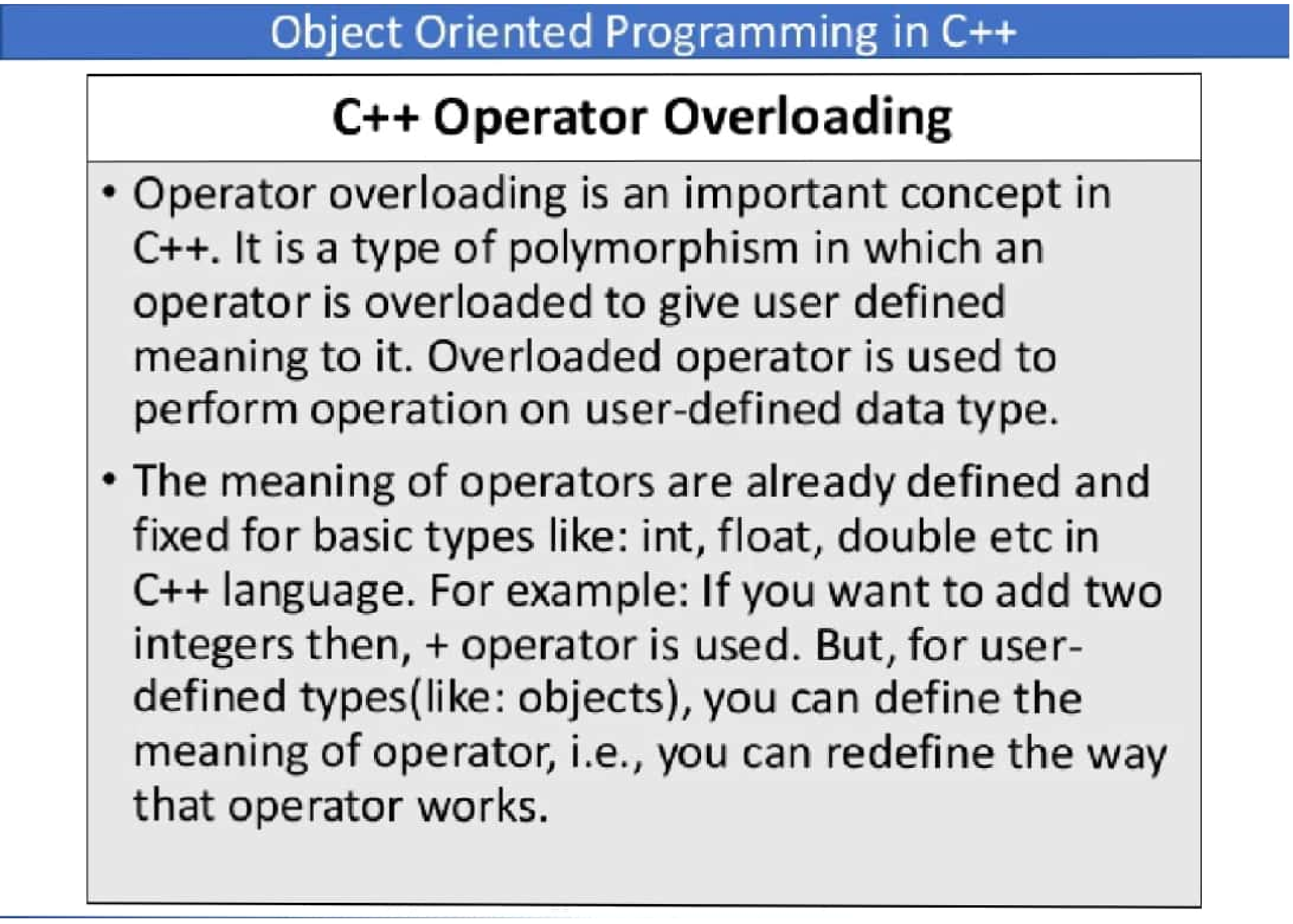
Function Call Binding using Base class Pointer

```
class Base {
 public:
 void show() {
            cout << "Base Class";   }
};
class Derived : public Base {
 public:
 void show() {
            cout << "Derived Class"; }
};
int main() {
 Base* b;       //Base class pointer
 Derived d;    //Derived class object
 b = &d;
 b->show();    //Early Binding Occurs
}
```

In the example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding. Compiler on seeing **Base class's pointer**, set call to Base class's **show()** function, without knowing the actual object type.

```
Base Class
```

& operator use for reference



C++ Operator Overloading

- Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type.
- The meaning of operators are already defined and fixed for basic types like: int, float, double etc in C++ language. For example: If you want to add two integers then, + operator is used. But, for user-defined types(like: objects), you can define the meaning of operator, i.e., you can redefine the way that operator works.

C++ Operator Overloading

- **For Example:** If there are two objects of a class that contain string as its data member, you can use + operator to concatenate two strings. Suppose, instead of strings if that class contains integer data member, then you can use + operator to add integers.

- This feature in C++ programming that allows programmer to redefine the meaning of operator when they operate on class objects is known as operator overloading.

C++ Operator Overloading

- **Why Operator overloading is used in C++.**
- In C++, whatever we can do by overloading an operator can be done without operator overloading. But operator overloading is used because it makes program more readable as the operator which are used for basic data types can also be used for user-defined data types. For example, consider a program that adds two complex number. To achieve this, we can create a friend function named add() that adds two complex number and return the result. We can call this function as,

```
c = add(c1 , c2);
```

- Here, c1 and c2 are two complex number to be added and c holds the result returned by the function. c, c1 and c2 are objects of a class complex. Using operator overloading, we can replace the calling statement as,

```
c = c1 + c2;
```

C++ Operator Overloading

- **Why Operator overloading is used in C++.**
- This statement gives more sense and user can clearly understand that two complex numbers are being added. Further statements like

```
z = add(mult(a,b) , sub(x,y));
```

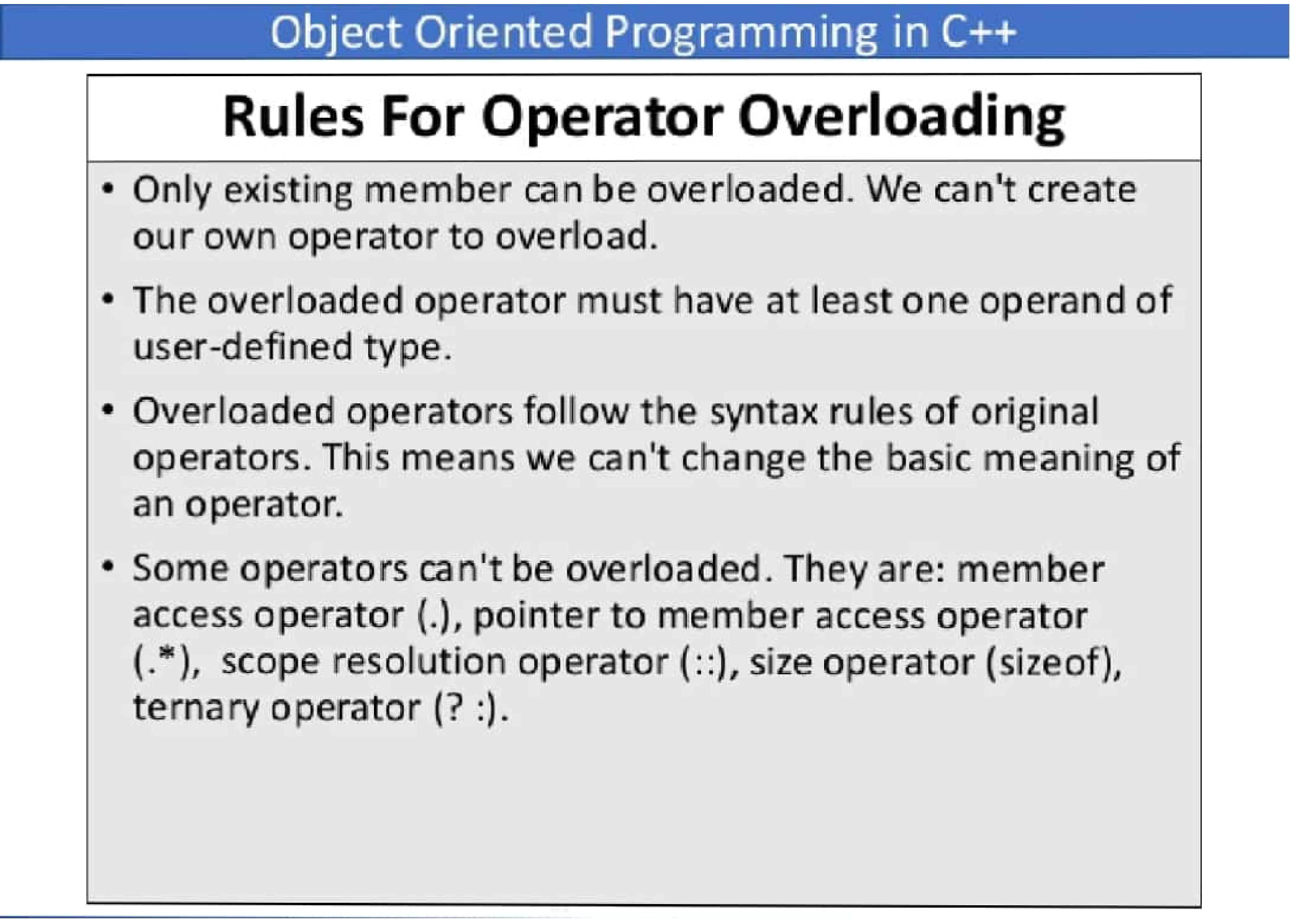
- can be replaced by

```
z = (a*b) + (x-y);
```

Object Oriented Programming in C++

```
class class_name
{
........ ...... ......
public:
   return_type operator sign (argument/s)
   {
    ...... ..... ......
   }
........ ...... ......
};
```

Operator function must be either friend function or non static member function. If the operator function is a friend function then it will have one argument for unary operator and two argument for binary operator. If the operator function is a non static member function then it will have no arguments for unary operators and one argument for binary operators.

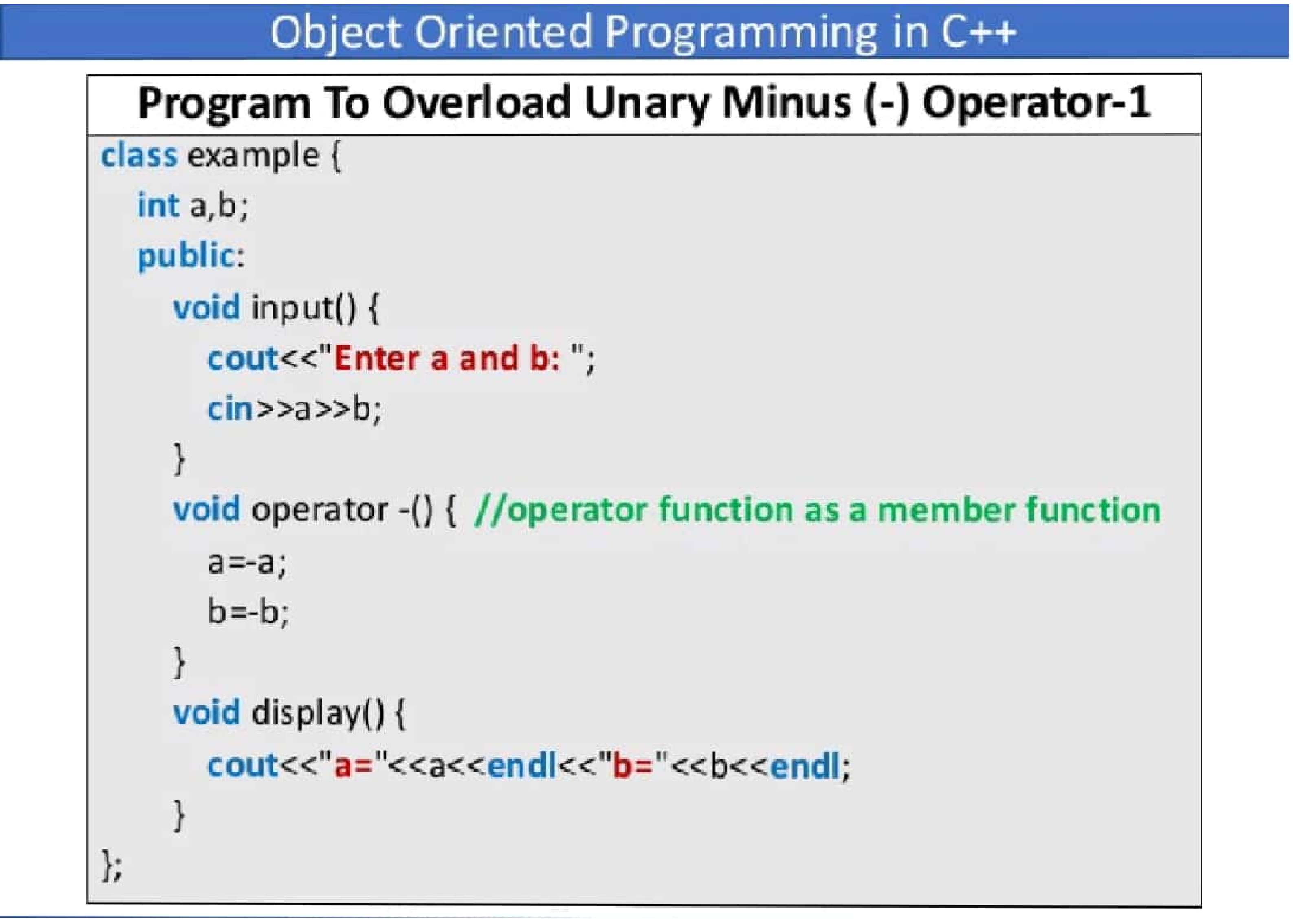


Rules For Operator Overloading

- Only existing member can be overloaded. We can't create our own operator to overload.
- The overloaded operator must have at least one operand of user-defined type.
- Overloaded operators follow the syntax rules of original operators. This means we can't change the basic meaning of an operator.
- Some operators can't be overloaded. They are: member access operator (.), pointer to member access operator (.*), scope resolution operator (::), size operator (sizeof), ternary operator (? :).

Rules For Operator Overloading

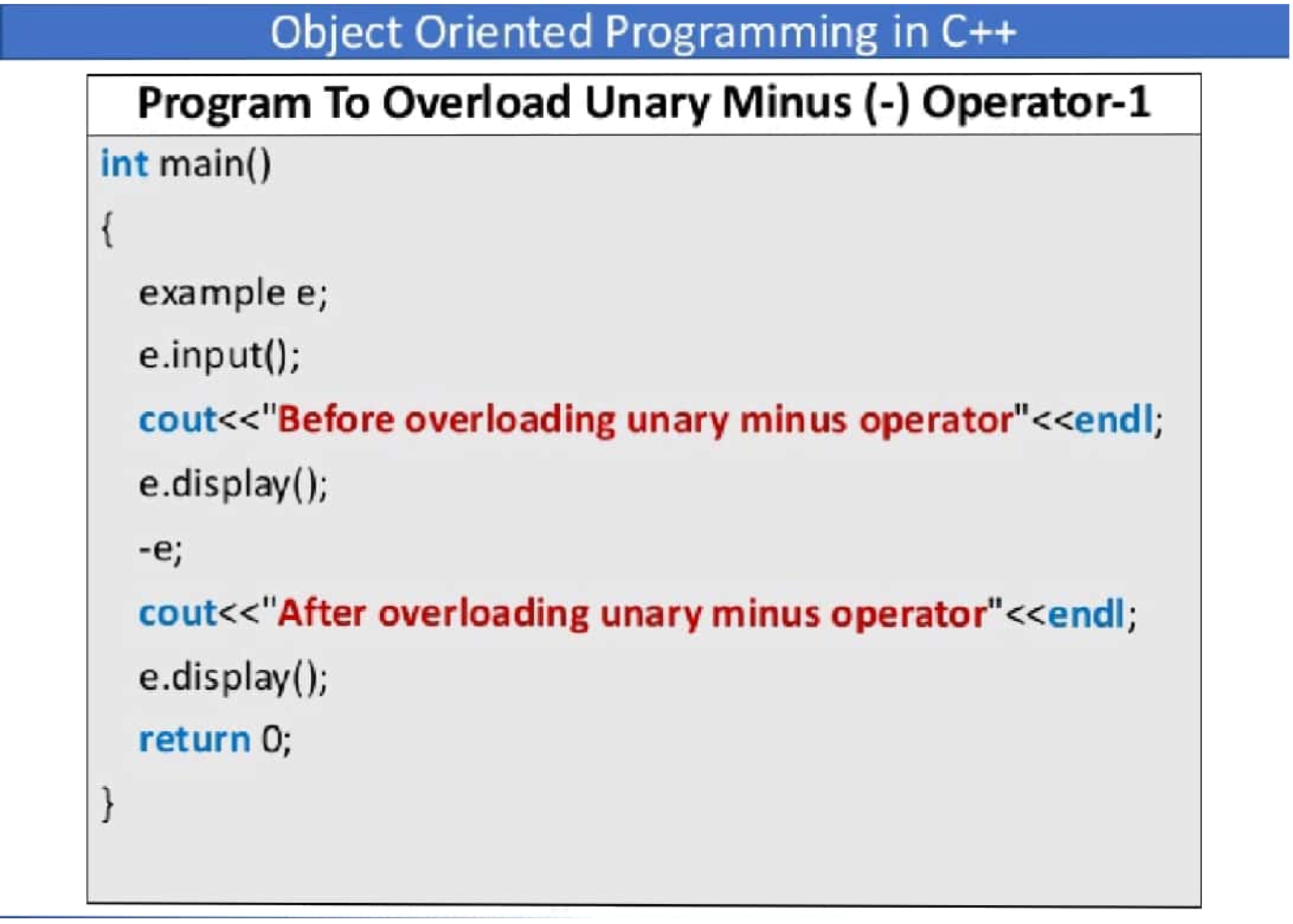
- We can't use friend function to overload some operators. They are: assignment operator (=), function call operator (()), subscripting operator ([]), class member access operator (->).
- If the operator function is a friend function then it will have one argument for unary operator and two argument for binary operator. If the operator function is a non static member function then it will have no arguments for unary operators and one argument for binary operators.
- When binary operators are overloaded through member function, the left hand operand must be an object of the relevant class.
- Binary arithmetic operators such as +, -, *, / must explicitly return a value.



Object Oriented Programming in C++

Program To Overload Unary Minus (-) Operator-1

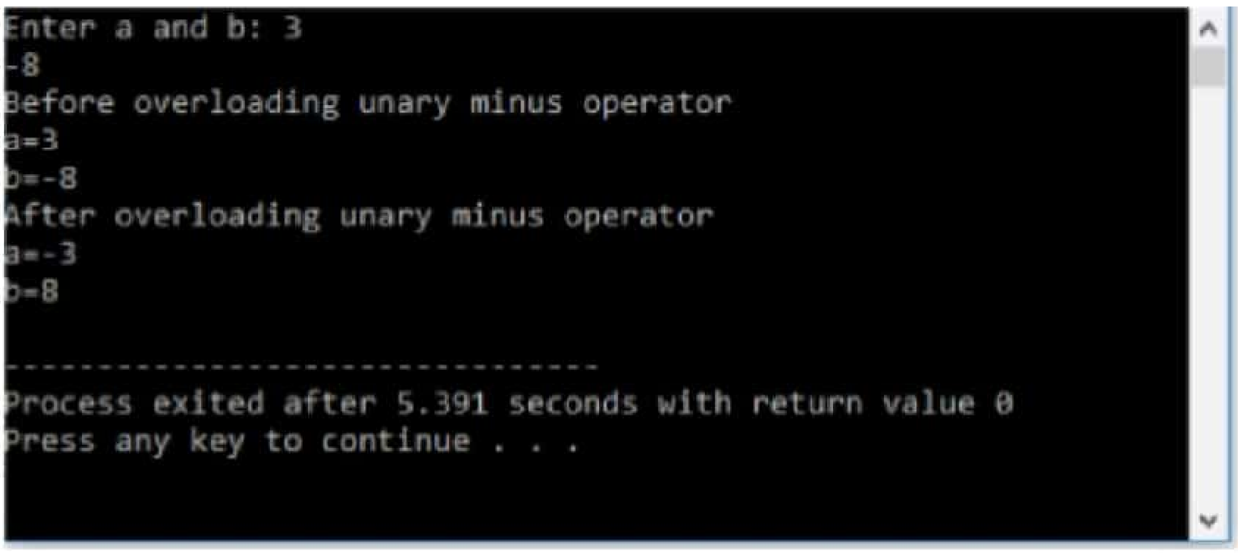
```
class example {
   int a,b;
   public:
      void input() {
         cout<<"Enter a and b: ";
         cin>>a>>b;
      }
      void operator -() {  //operator function as a member function
         a=-a;
         b=-b;
      }
      void display() {
         cout<<"a="<<a<<endl<<"b="<<b<<endl;
      }
};
```



Object Oriented Programming in C++

Program To Overload Unary Minus (-) Operator-1

```
int main()
{
   example e;
   e.input();
   cout<<"Before overloading unary minus operator"<<endl;
   e.display();
   -e;
   cout<<"After overloading unary minus operator"<<endl;
   e.display();
   return 0;
}
```



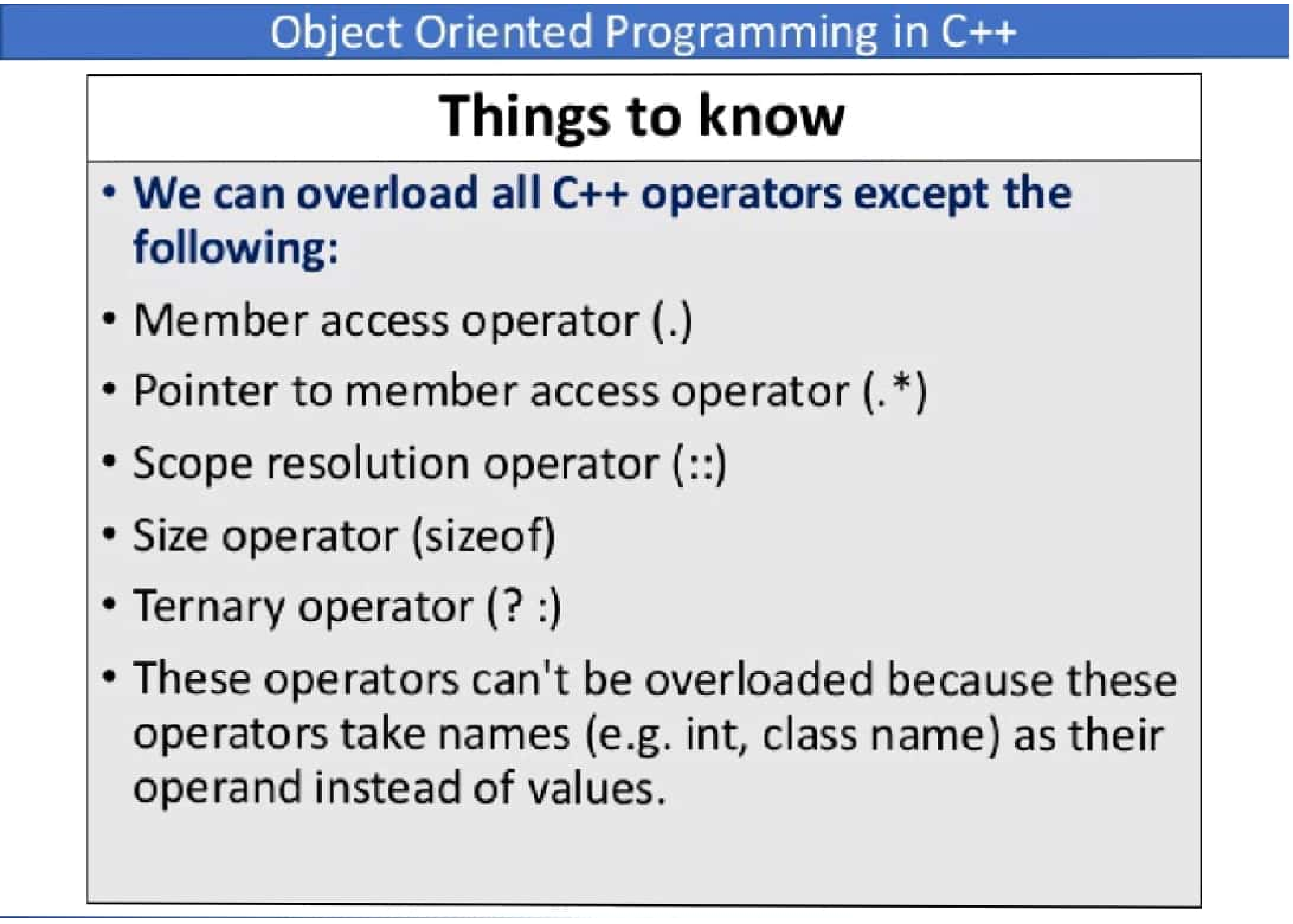
```
Enter a and b: 3
-8
Before overloading unary minus operator
a=3
b=-8
After overloading unary minus operator
a=-3
b=8

--------------------------------
Process exited after 5.391 seconds with return value 0
Press any key to continue . . .
```

To Overload Unary Minus (-) Operator

- Friend function can also be used as operator function as:

```
friend void operator -(example s)
{
   s.a=-s.a;
   s.b=-s.b;
}
```

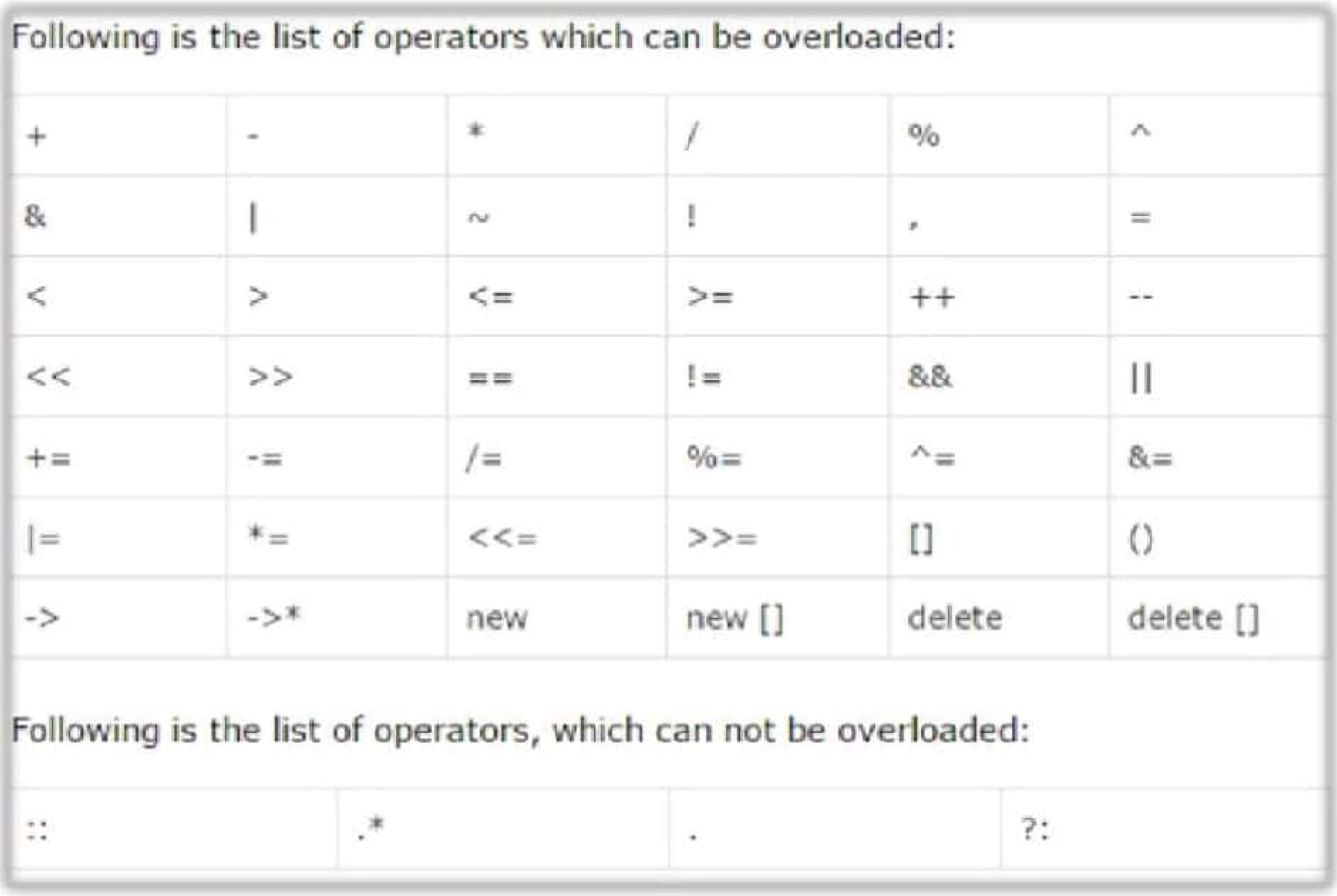
- This program shows how to overload unary minus operator. As per the rule, if non static member function is used, it will take no argument and if a friend function is used as operator function, one argument is required. The statement ' *-e;* ' calls the operator function. If the operator function is friend function then the object *e* is the argument itself. Inside the operator function, the sign of data is changed. Hence, unary minus operator is overloaded.



Things to know

- **We can overload all C++ operators except the following:**
- Member access operator (.)
- Pointer to member access operator (.*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Ternary operator (? :)
- These operators can't be overloaded because these operators take names (e.g. int, class name) as their operand instead of values.

Overloadable/Non Overloadable Operators

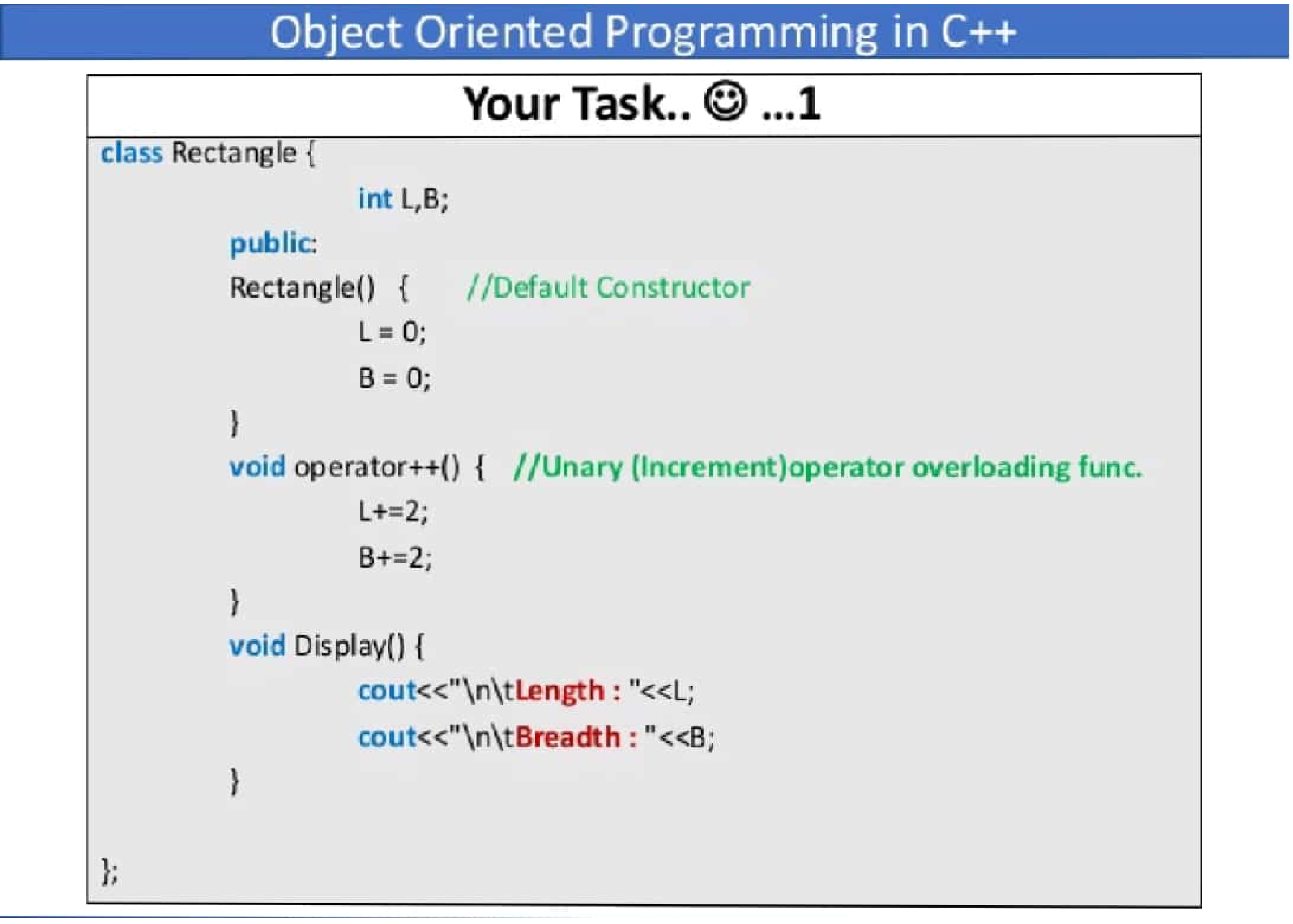


Following is the list of operators which can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded:

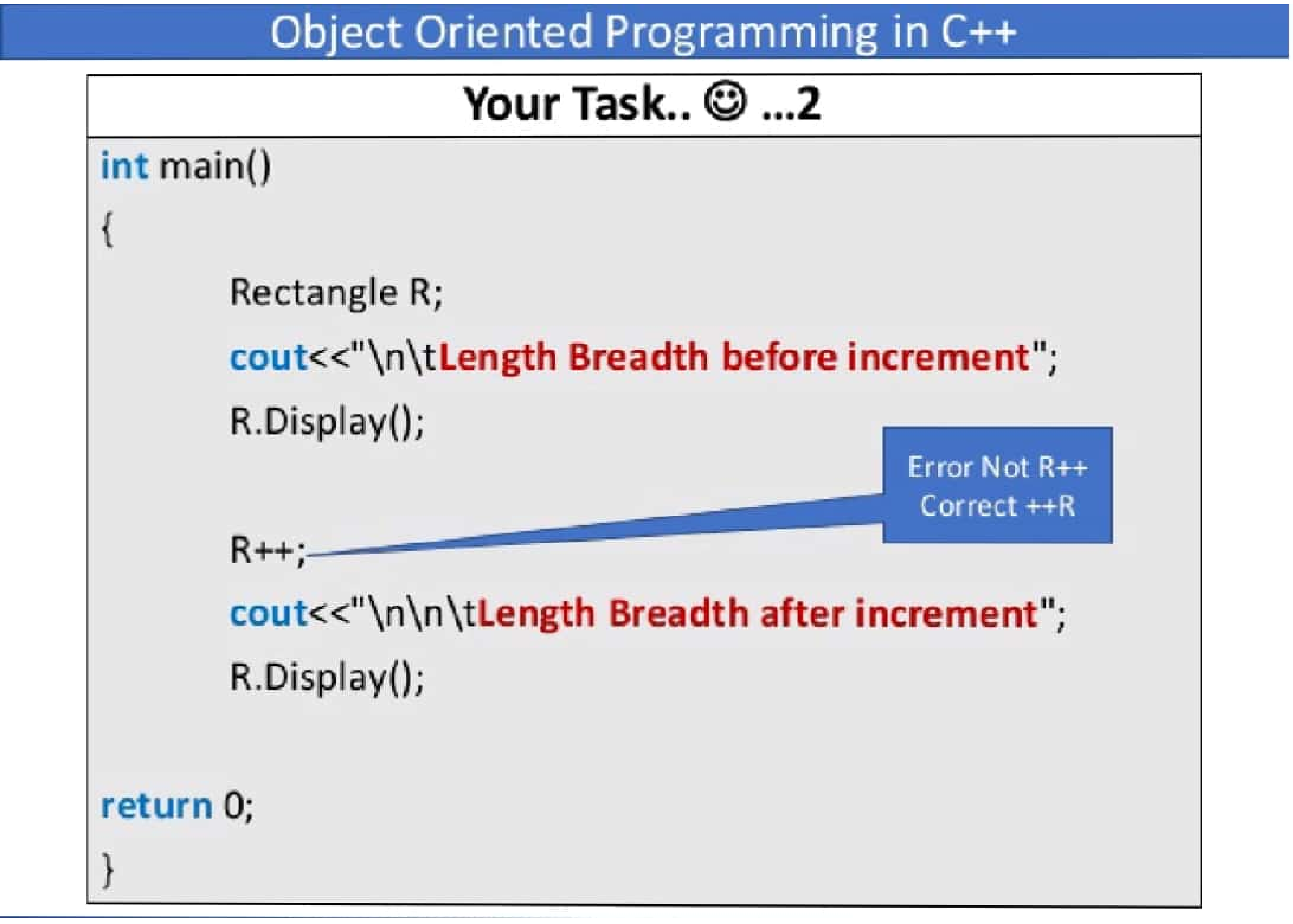
::	.*	.	?:



Object Oriented Programming in C++

Your Task.. ☺ ...1

```
class Rectangle {
                int L,B;
        public:
        Rectangle()  {      //Default Constructor
                L = 0;
                B = 0;
        }
        void operator++() {   //Unary (Increment)operator overloading func.
                L+=2;
                B+=2;
        }
        void Display() {
                cout<<"\n\tLength : "<<L;
                cout<<"\n\tBreadth : "<<B;
        }

};
```

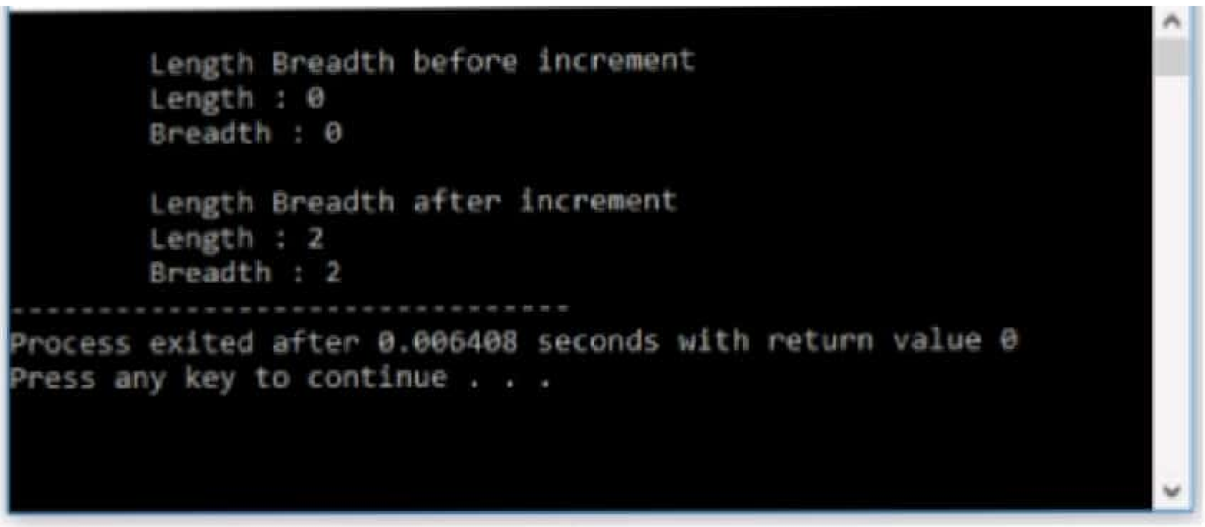


Object Oriented Programming in C++

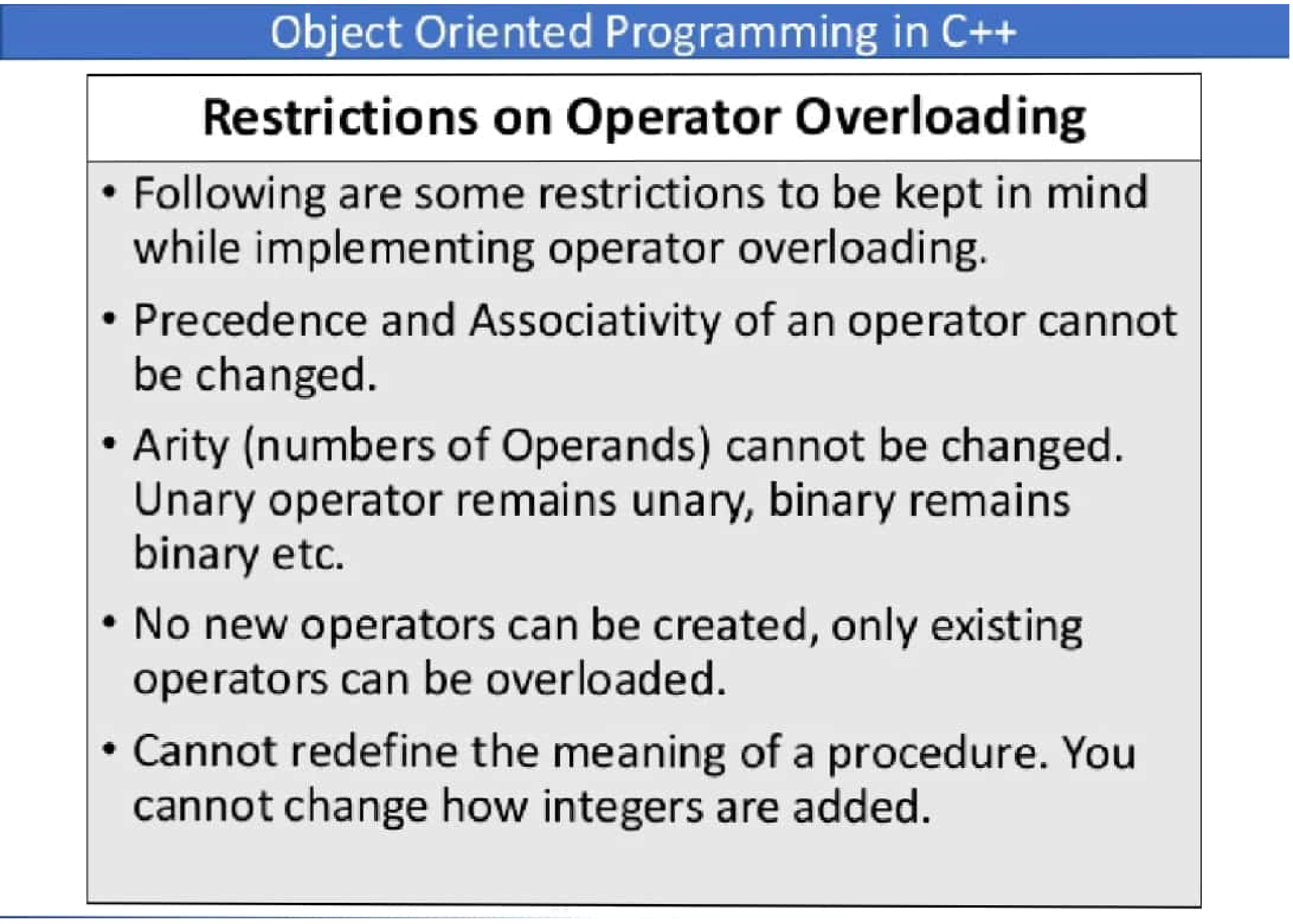
Your Task.. ☺ ...2

```
int main()
{
        Rectangle R;
        cout<<"\n\tLength Breadth before increment";
        R.Display();

        R++;
        cout<<"\n\n\tLength Breadth after increment";
        R.Display();


return 0;
}
```

Error Not R++
Correct ++R



```
        Length Breadth before increment
        Length : 0
        Breadth : 0

        Length Breadth after increment
        Length : 2
        Breadth : 2
--------------------------------
Process exited after 0.006408 seconds with return value 0
Press any key to continue . . .
```



Restrictions on Operator Overloading

- Following are some restrictions to be kept in mind while implementing operator overloading.
- Precedence and Associativity of an operator cannot be changed.
- Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
- No new operators can be created, only existing operators can be overloaded.
- Cannot redefine the meaning of a procedure. You cannot change how integers are added.