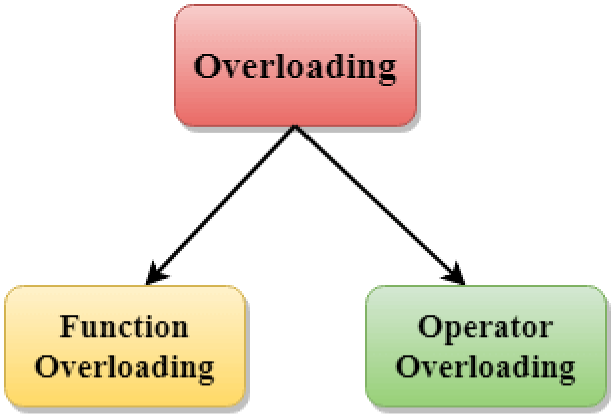
C++ Overloading (Function and Operator)

Definition

- If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:
- methods
- constructors
- indexed properties
- It is because these members have parameters only.

Types of overloading in C++

- Function overloading
- Operator overloading



Overloading

Function Overloading

Operator Overloading

C++ Function Overloading

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.
- In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.
- The **advantage** of Function overloading is that it increases the readability of the program because we don't need to use different names for the same action.

Example 1

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                                          //     class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

Example 2

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   cout << "r1 is : " <<r1<< endl;
   cout <<"r2 is : "  <<r2<< endl;
   return 0;
}
```

Function Overloading and Ambiguity

- When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.
- When the compiler shows the ambiguity error, the compiler does not run the program.
- **Causes of Function Overloading:**
- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

Type Conversion

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   cout << "Value of i is : " <<i<< endl;
}
void fun(float j)
{
   cout << "Value of j is : " <<j<< endl;
}
int main()
{
   fun(10);
   fun(1.2);
   return 0;
}
```

- The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Operator Overloading

- In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.
- Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
  Example:
-
  ```
   int a;
  float b, sum;
  sum=a+b;
  ```
- Here, variables “a” and “b” are of types “int” and “float”, which are built-in data types. Hence the addition operator ‘+’ can easily add the contents of “a” and “b”. This is because the addition operator “+” is predefined to add variables of built-in data type only.

Now, consider another example

```
class A
    {
    };
int main()
    {
        A   a1,a2,a3;
       a3= a1 + a2;
       return 0;
    }
```

- In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

- In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,
- Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.
- Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

  ```
  result = c1 + c2;
  ```
- instead of something like:

  ```
  result = c1.addNumbers(c2);
  ```
- **Syntax for C++ Operator Overloading**

  To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

  ```
  returnType operator symbol (arguments) { }
  ```

Operator Overloading in Unary Operators

- Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

```
class Count {
private:
int value;
public:
// Constructor to initialize count to 5
Count() : value(5) {}
// Overload ++ when used as prefix
void operator ++ () {
++value; }
void display() {
cout << "Count: " << value << endl;
} };
int main() {
Count count1;
 // Call the "void operator ++ ()" function
++count1;
count1.display();
return 0; }
```

Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

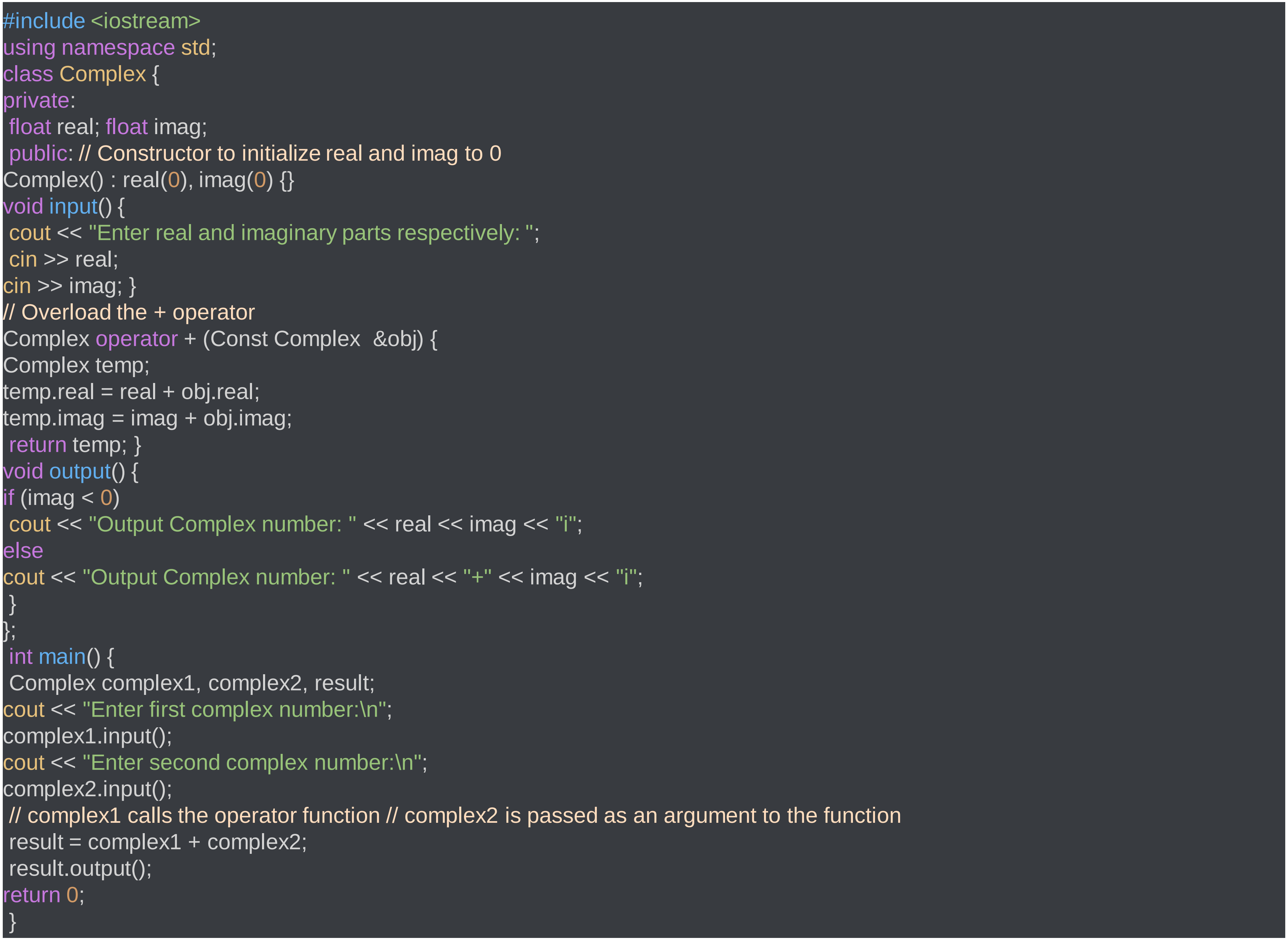
- sizeof
- typeid
- Scope resolution (::)
- Class member access operators (.(dot), .* (pointer to member operator))
- Ternary or conditional (?:)

Binary operator Overloading

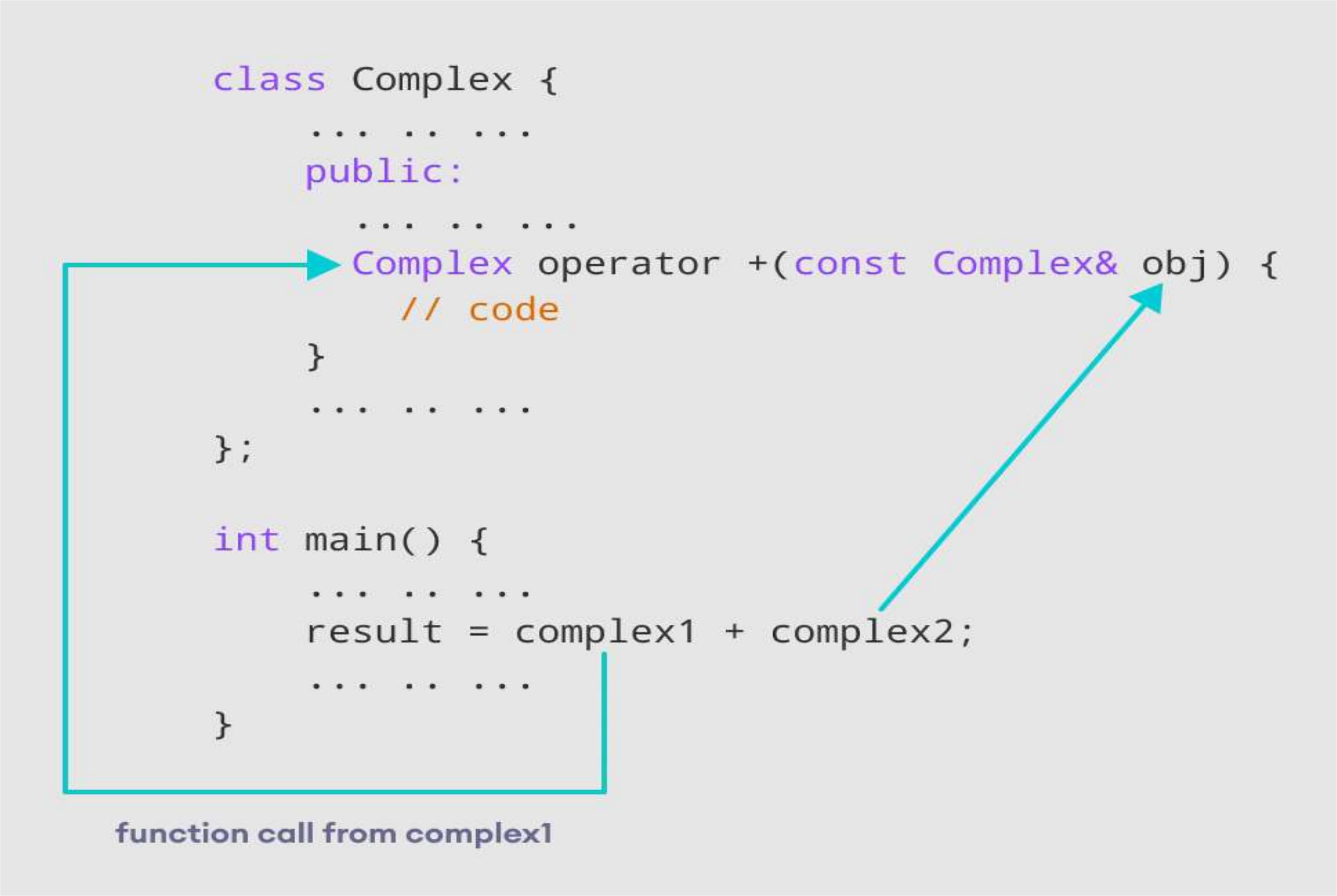
```
#include <iostream>
using namespace std;
class Complex_num
{
    int x, y;
    public:
        void input()    {
            cout << " Input two complex number: " << endl;
            cin >> x >> y;
        }
        // use binary '+' operator to overload
        Complex_num operator + (Complex_num  &obj)
        {
            // create an object
            Complex_num A;
            // assign values to object
            A.x = x + obj.x;
            A.y = y + obj.y;
            return (A);
        }
        // overload the binary (-) operator
Complex_num operator - (Complex_num  &obj)
        {
            Complex_num A;
            // assign values to object
            A.x = x - obj.x;
            A.y = y - obj.y;
            return (A);
        }
```

```
 // display the result of addition
      void print1()  {
         cout << x << " + " << y << "i" << "\n";  }
// display the result of subtraction
      void print2()  {
         cout << x << " - " << y << "i" << "\n";  }
};
int main ()  {
Complex_num x1, y1, sum, sub;
                  // here we created object of class Addition i.e x1 and y1  accepting the values
   x1.input();
   y1.input();
   sum = x1 + y1;                              // add the objects
   sub = x1 - y1;                             // subtract the complex number
   // display user entered values
   cout << "\n Entered values are: \n";
   cout << " \t";
   x1.print1();
   cout << " \t";
   y1.print1();
   cout << "\n The addition of two complex (real and imaginary) numbers: ";
   sum.print1(); // call print function to display the result of addition
   cout << "\n The subtraction of two complex (real and imaginary) numbers: ";
   sub.print2(); // call print2 function to display the result of subtraction
   return 0;  }
```

Addition of two Complex Numbers



```
#include <iostream>
using namespace std;
class Complex {
private:
 float real; float imag;
 public: // Constructor to initialize real and imag to 0
Complex() : real(0), imag(0) {}
void input() {
 cout << "Enter real and imaginary parts respectively: ";
 cin >> real;
cin >> imag; }
// Overload the + operator
Complex operator + (Const Complex  &obj) {
Complex temp;
temp.real = real + obj.real;
temp.imag = imag + obj.imag;
 return temp; }
void output() {
if (imag < 0)
 cout << "Output Complex number: " << real << imag << "i";
else
cout << "Output Complex number: " << real << "+" << imag << "i";
 }
};
 int main() {
 Complex complex1, complex2, result;
cout << "Enter first complex number:\n";
complex1.input();
cout << "Enter second complex number:\n";
complex2.input();
 // complex1 calls the operator function // complex2 is passed as an argument to the function
 result = complex1 + complex2;
 result.output();
return 0;
 }
```



```
class Complex {
    ... .. ...
    public:
      ... .. ...
      Complex operator +(const Complex& obj) {
        // code
    }
    ... .. ...
};

int main() {
    ... .. ...
    result = complex1 + complex2;
    ... .. ...
}
```