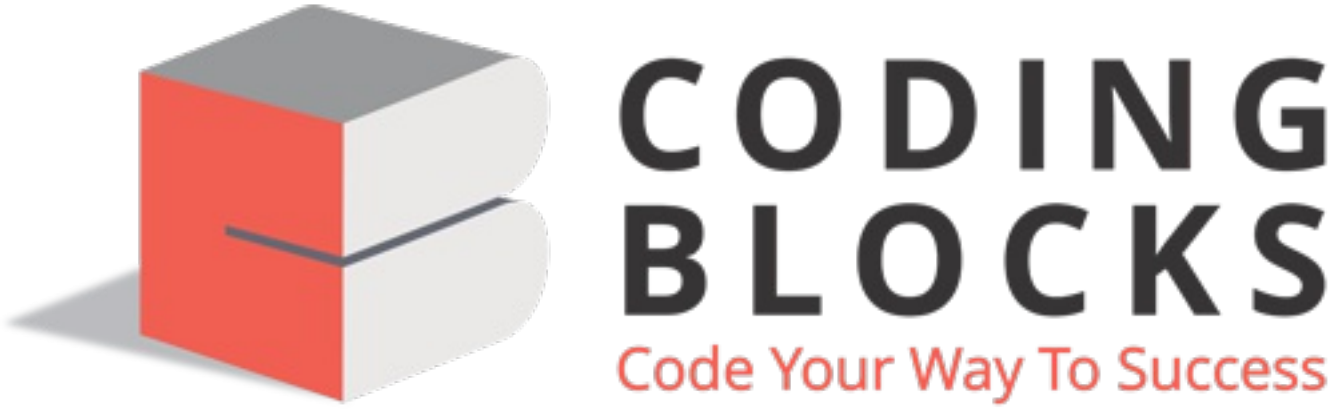

Object Oriented Programming - Function Overloading and Operator Overloading

Function Overloading

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Example

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
```

```
void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

OUTPUT

```
Here is int 10
Here is float 10.1
Here is char* ten
```

Functions that cannot be overloaded in C++

- Function declarations that differ only in the return type.
- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.
- Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to

become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

Operator Overloading

- Operators that operate on two operands are known as **binary** operators.
- Operators that operate on one operand are known as **unary** operators.

Example to Add Two Complex Numbers Using Add Function

Run on IDE

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

class Complex{
    int real;
    int img;
public:
    Complex(){
```

```
        real = 0;
        img = 0;
    }

    Complex(int r,int i){
        real = r;
        img = i;
    }

    void setReal(const int r){
        real = r;
    }

    void setImg(const int i){
        img = i;
    }

    int getReal() const{
        return real;
    }

    int getImg() const{
        return img;
    }

    void print(){
        if(img>0){
            cout<<real<<" + "<<img<<"i"<<endl;
```

```
        }else{
            cout<<real<<" - "<<img<<"i"<<endl;
        }
    }

    void add(const Complex &x){
        real +=x.real;
        img +=x.img;
    }
};

int main() {
    Complex c1(5,3);
    Complex c2;
    c2.setReal(1);
    c2.setImg(4);

    c1.print();
    c2.print();

    c1.add(c2);
    c1.print();
    c2.print();

    return 0;
}
```

OUTPUT

```
5 + 3i
1 + 4i
6 + 7i
1 + 4i
```

Overloading “+” Operator

```
void operator+(const Complex &x){
     real +=x.real;
    img +=x.img;
}
```

Overloading “!” Operator

```
void operator!(){
    img *= -1;
}
```

Overloading []

```
int operator[](string s){
    if(s=="real")
        return real;
    else
```

```
        return img;
}
```

Overloading >> and <<

```
void operator>>(istream& is,Complex &c){
    int r1,i1;
    is>>r1>>i1;
    c.setReal(r1);
    c.setImg(i1);
}

void operator<<(ostream& is,Complex &c){
    c.print();
}
```

Cascading Operators

The multiple use of input or output operators (“>>”or”<<”) in one statement is called cascading of I/O operator.

Note The above code works fine to input one object, Complex number in this case but this code will fail if you try to input or output two complex numbers. To be able to input two complex numbers, make return type of the above functions to **istream&** and **ostream&** respectively and return **is** and **os** respectively.

Run on IDE

```
istream& operator>>(istream& is,Complex &c){
    int r1,i1;
    is>>r1>>i1;
    c.setReal(r1);
    c.setImg(i1);
    return is;
}

ostream& operator<<(ostream& os,Complex &c){
    c.print();
    return os;
}
```

Example to Add Two Complex Numbers Using Operator Overloading

Run on IDE

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

class Complex{
    int real;
    int img;
public:
    Complex(){
```

```
        real = 0;
        img = 0;
    }

    Complex(int r,int i){
        real = r;
        img = i;
    }

    void setReal(const int r){
        real = r;
    }

    void setImg(const int i){
        img = i;
    }

    int getReal() const{
        return real;
    }

    int getImg() const{
        return img;
    }

    void print(){
        if(img>0){
            cout<<real<<" + "<<img<<"i"<<endl;
```

```
        }else{
            cout<<real<<" - "<<-img<<"i"<<endl;
        }
    }

    void add(const Complex &x){
        real +=x.real;
        img +=x.img;
    }

    void operator+(const Complex &x){
         real +=x.real;
        img +=x.img;
    }

    void operator!(){
        img *= -1;
    }

    int operator[](string s){
        if(s=="real")
            return real;
        else
            return img;
    }
};
/*
void operator>>(istream& is,Complex &c){
```

```
    int r1,i1;
    is>>r1>>i1;
    c.setReal(r1);
    c.setImg(i1);
}

void operator<<(ostream& is,Complex &c){
    c.print();
}
*/
istream& operator>>(istream& is,Complex &c){
    int r1,i1;
    is>>r1>>i1;
    c.setReal(r1);
    c.setImg(i1);
    return is;
}

ostream& operator<<(ostream& os,Complex &c){
    c.print();
    return os;
}

int main() {
    Complex c1(5,3);
    Complex c2;
    c2.setReal(1);
    c2.setImg(4);
```

```
    c1.print();
    c2.print();

    //c1.add(c2);
    c1 + c2;
    c1.print();

    !c1;
    c1.print();

    cout<<c1["img"];
    return 0;
}
```