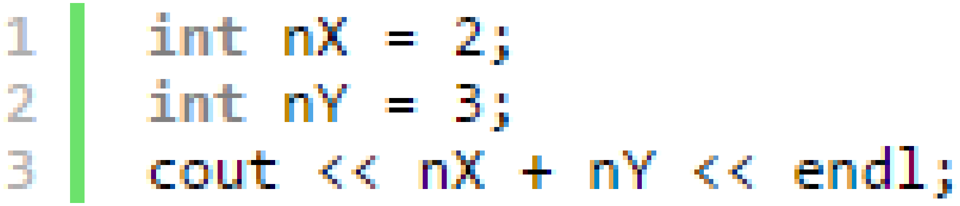
OPERATOR OVERLOADING

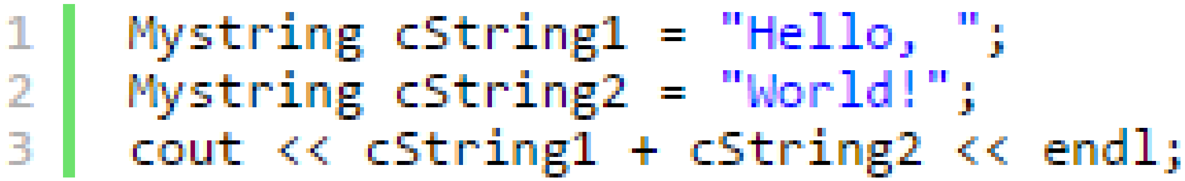
Operator Overloading



```
int nX = 2;
int nY = 3;
cout << nX + nY << endl;
```

C++ already knows how the plus operator (+) should be applied to integer operands — the compiler adds nX and nY together and returns the result.

Operator Overloading



```
Mystring cString1 = "Hello, ";
Mystring cString2 = "World!";
cout << cString1 + cString2 << endl;
```

- What would you expect to happen in this case.
- However, because Mystring is a user-defined class, C++ does not know what operator + should do.
- We need to tell it how the + operator should work with two objects of type Mystring.
- Once an operator has been overloaded, C++ will call the appropriate overloaded version of the operator based on parameter type.
- If you add two integers, the integer version of operator plus will be called.
- If you add two Mystrings, the Mystring version of operator plus will be called.

Operator Overloading

- Almost any operator in C++ can be overloaded. The exceptions are: arithmetic if (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*).
- At least one of the operands in any overloaded operator must be a user-defined type.
- Only the existing operator can be overloaded.
- All operators keep their current precedence and associativity, regardless of what they're used for.

Operator Overloading

- Cannot change
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order of operators
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - e.g., & is unary, can only act on one operand
  - How operators act on built-in data types (i.e., cannot change integer addition)

Defining Operator Overloading

- To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied.

- This is done with the help of a special function called *operator function.*

```
return type class-name : :operator op (arg-list)
{
    Function body   //  task defined
}
```

Defining Operator Overloading

```
return type class-name : :operator op (arg-list)
{
      Function body   //  task defined
}
```

- ***return type*** is the type of value returned by the specified operation.
- **op** is the operator being overloaded.
- **op** is preceded by the keyword **operator.**
- ***operator* op** is the function name.

Defining Operator Overloading
continue…

Operator Function must be either

- member function (non-static)

Or

- friend function.

The basic difference :

- A friend function will have only one argument for unary operators and two for binary operators.
- A member function has no arguments for unary operators and one argument for binary operators.
- This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function.
- Arguments may be passed either by value or by reference.

Process of Operator Overloading

The process of overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.

- Declare the operator function ***operator op( )*** in the public part of the class. It may be either a member function or a friend function.

- Define the operator function to implement the required operations.

Process of Operator Overloading

Overloaded operator functions can be invoked by expressions such as:

For unary operators: op x or x op

For binary operators: x op y

op x or x op would be interpreted as

for a friend function: operator op (x)

for a member function: x.operator op ( )

x op y would be interpreted as

for a friend function: operator op (x,y)

for a member function: x.operator op (y)

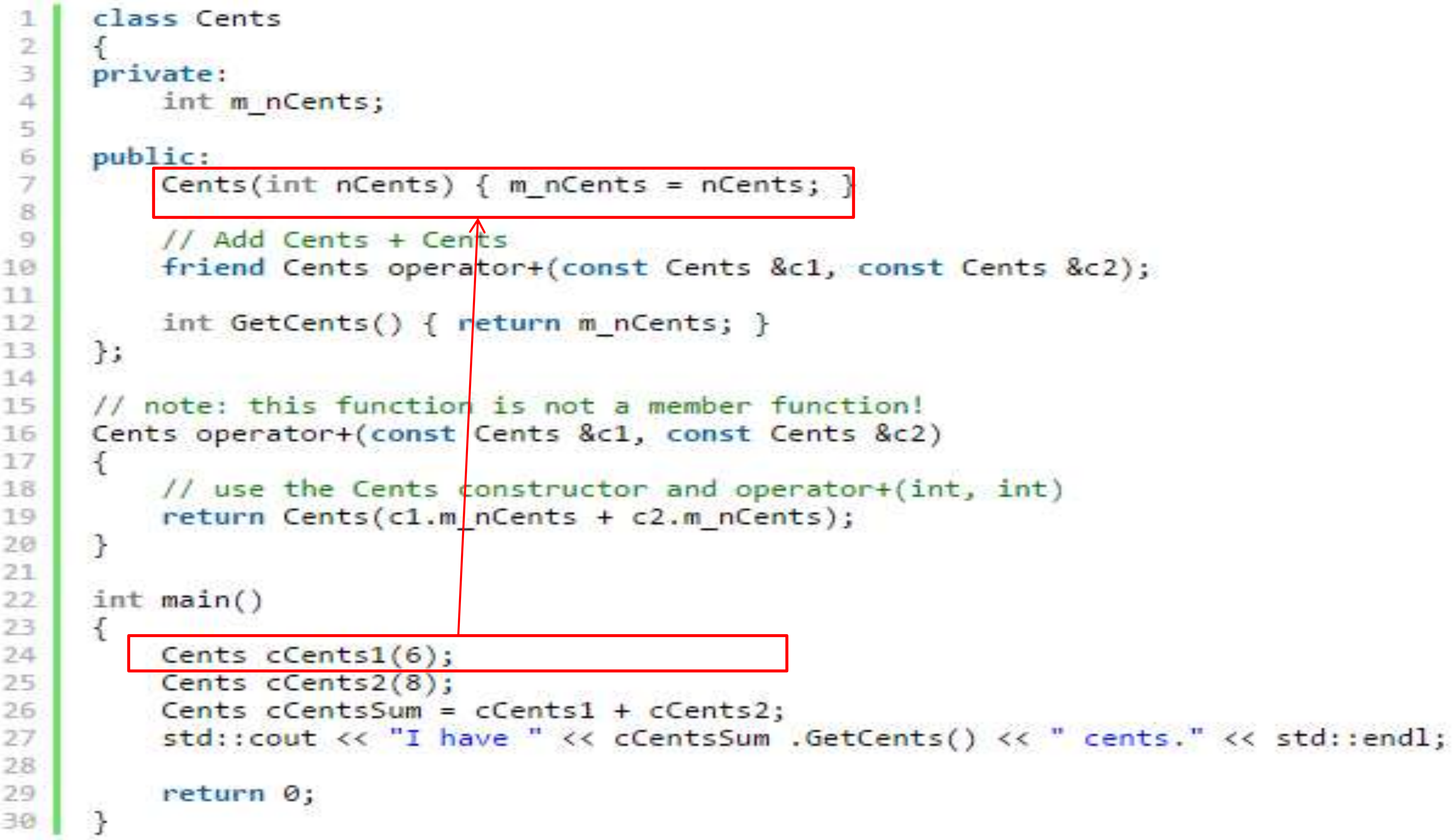
Operators as functions

- nX + nY: *operator+(nX, nY) (where operator+ is the name of the function).*
- Similarly dX + dY becomes *operator+(dX, dY).*
- Even though both expressions call a function named ***operator+(),*** function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).

Overloading the arithmetic operators using friend function

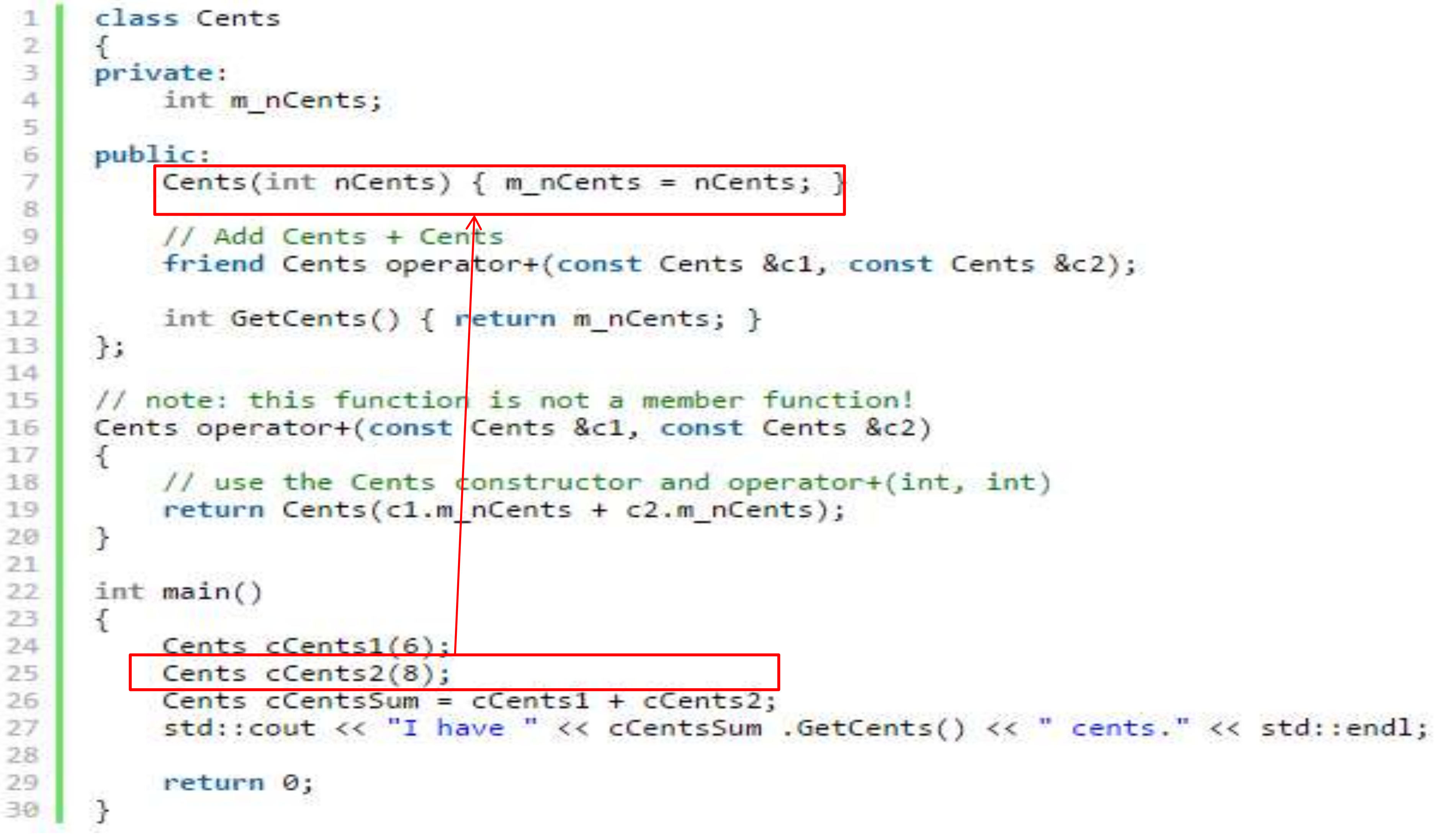
- When the operator does not modify its operands, the best way to overload the operator is via **friend function.**
- None of the arithmetic operators modify their operands (they just produce and return a result), so we will utilize the friend function overloaded operator method here.

Overload + operator



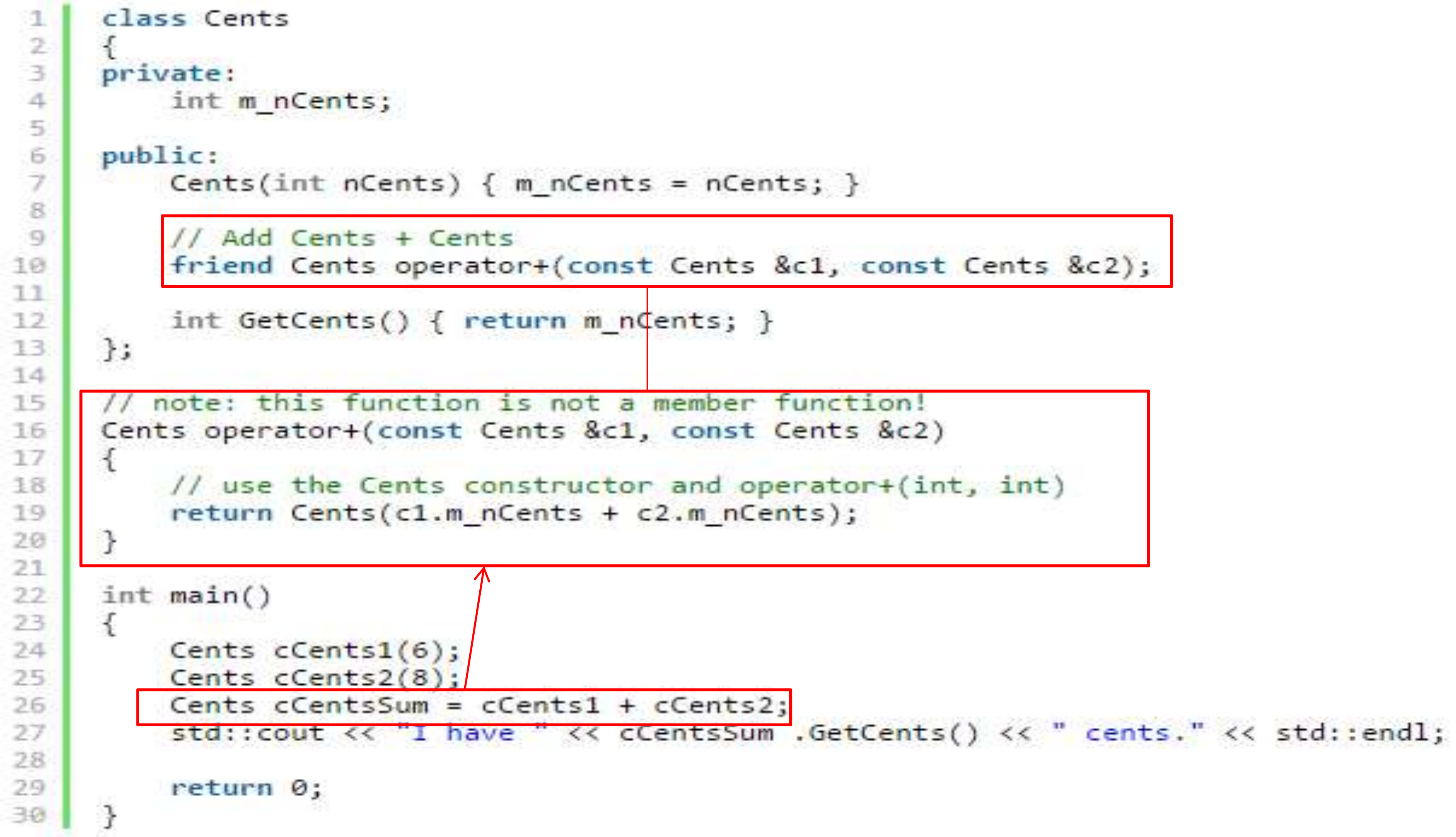
```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Add Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = cCents1 + cCents2;
    std::cout << "I have " << cCentsSum .GetCents() << " cents." << std::endl;

    return 0;
}
```

Overload + operator



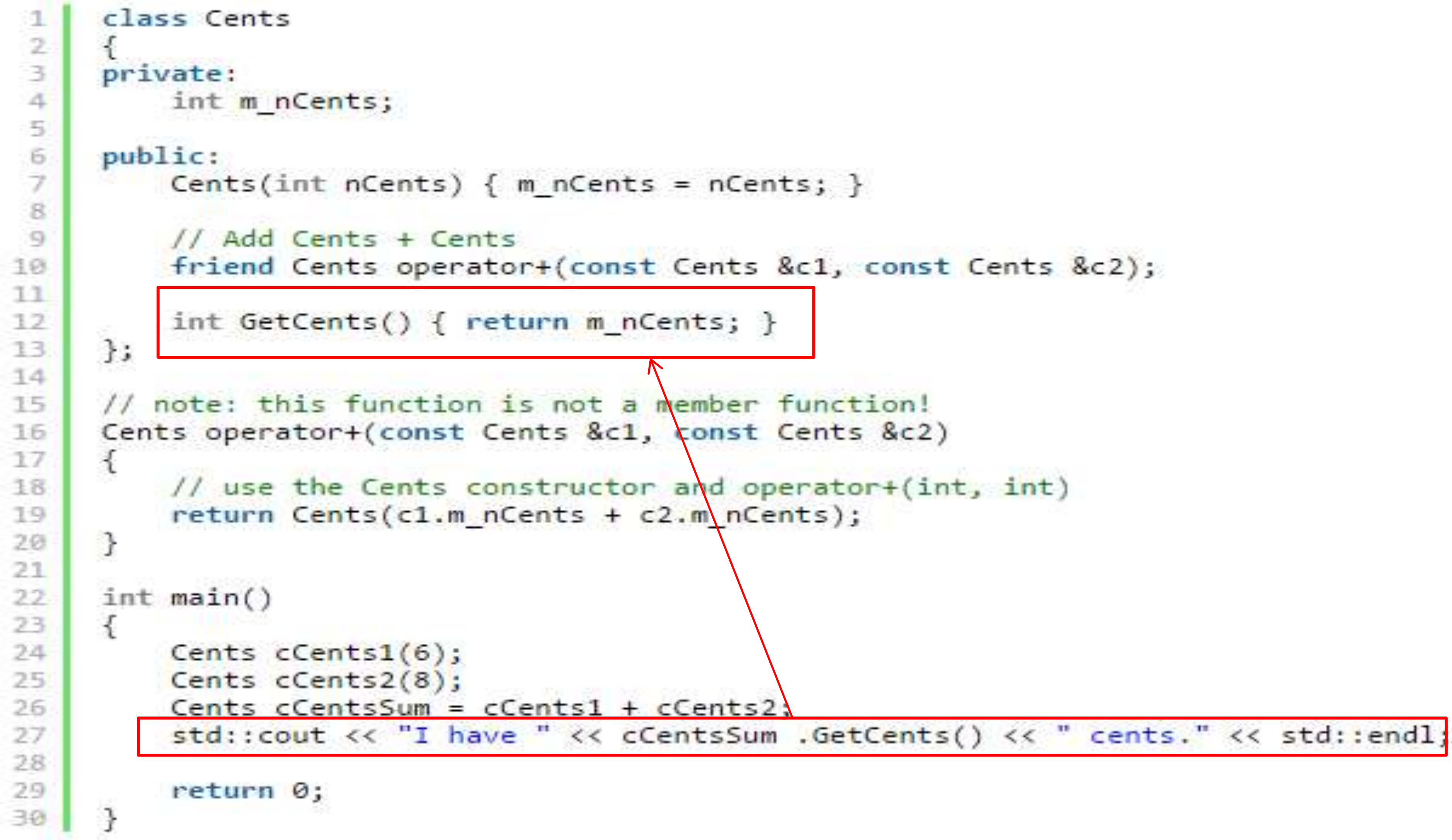
```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Add Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = cCents1 + cCents2;
    std::cout << "I have " << cCentsSum .GetCents() << " cents." << std::endl;

    return 0;
}
```

Overload + operator



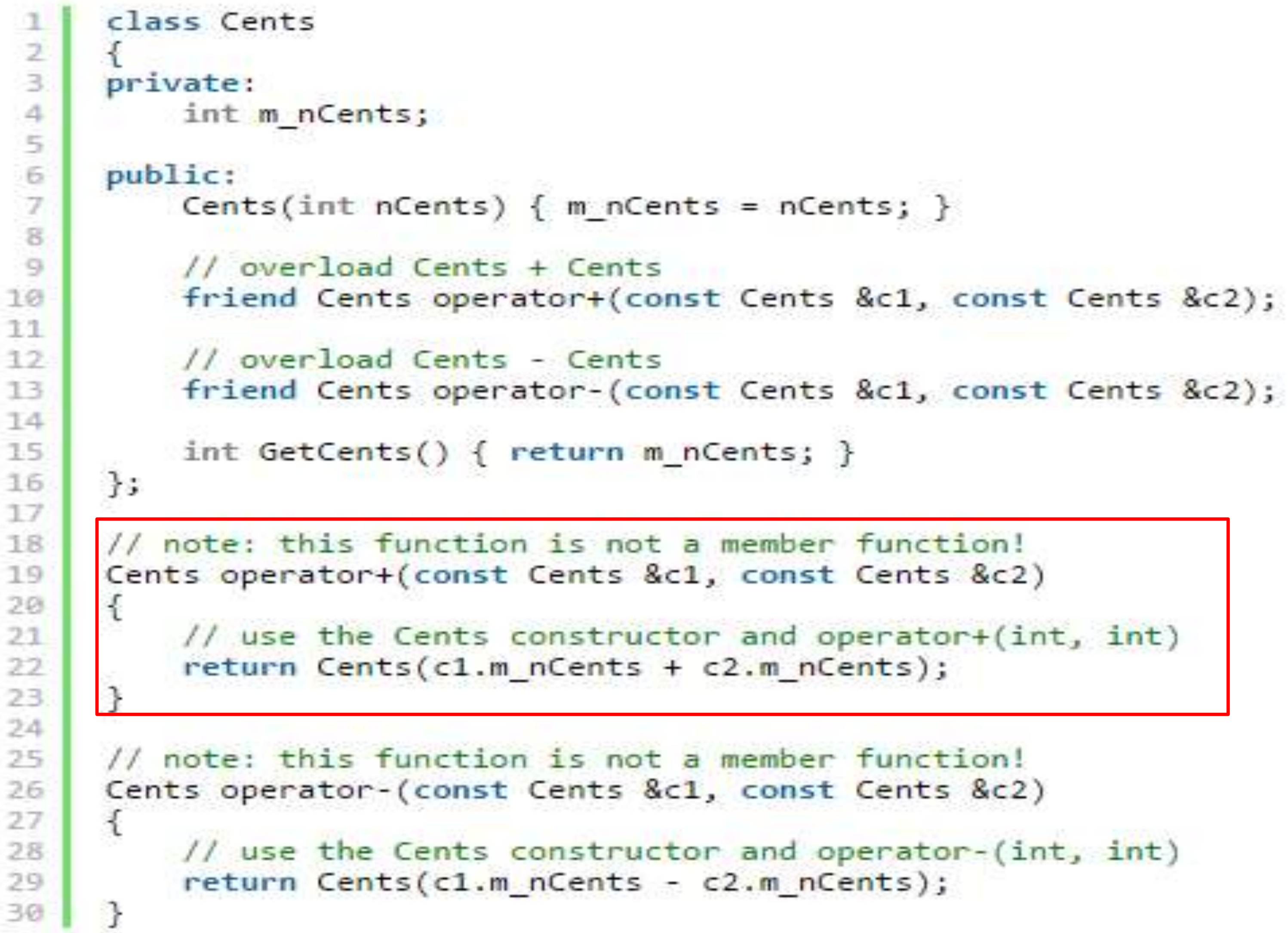
```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Add Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = cCents1 + cCents2;
    std::cout << "I have " << cCentsSum .GetCents() << " cents." << std::endl;

    return 0;
}
```

Overload + operator



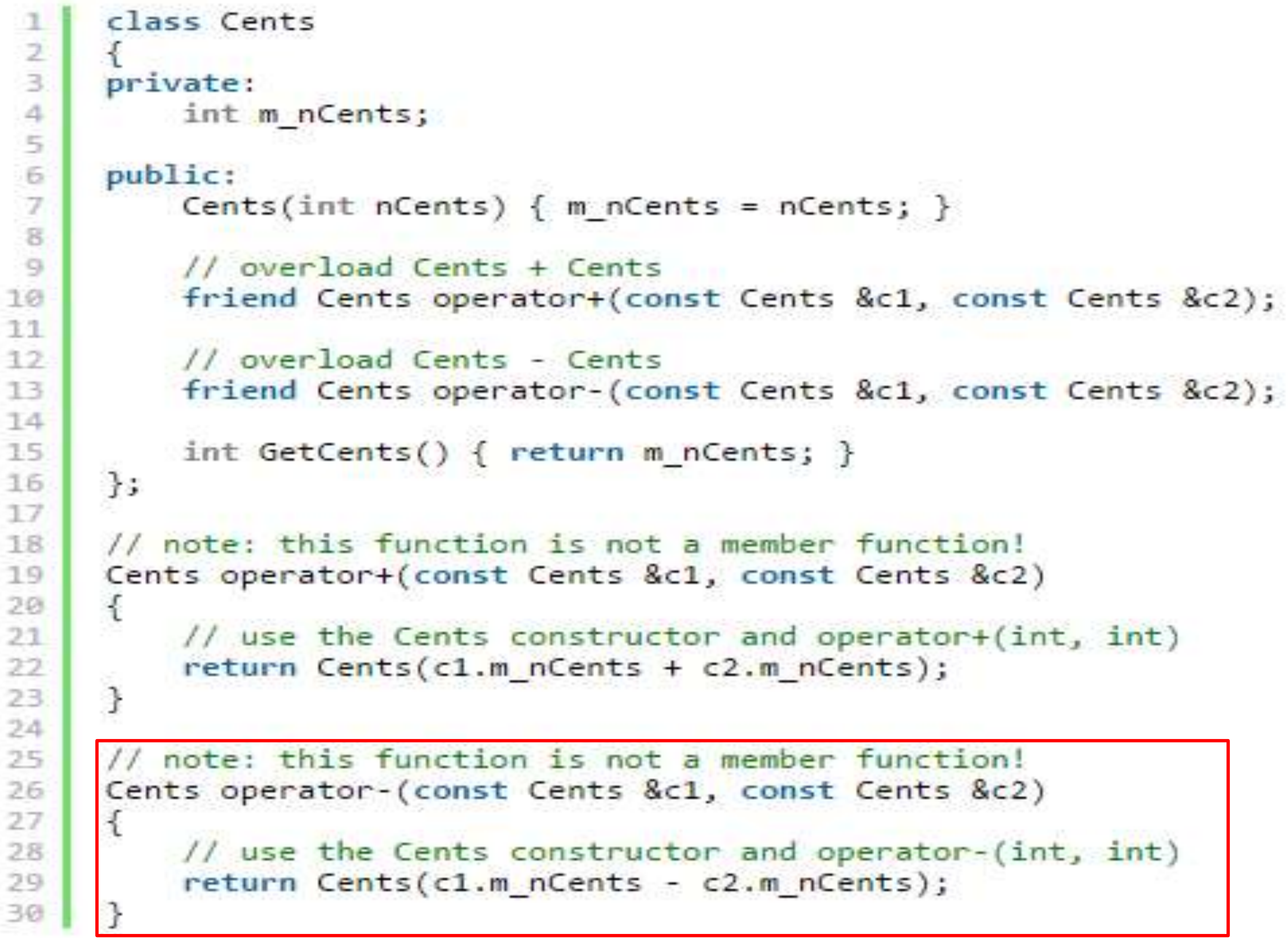
```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Add Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = cCents1 + cCents2;
    std::cout << "I have " << cCentsSum .GetCents() << " cents." << std::endl;

    return 0;
}
```

Overload - operator



```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // overload Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    // overload Cents - Cents
    friend Cents operator-(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

// note: this function is not a member function!
Cents operator-(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator-(int, int)
    return Cents(c1.m_nCents - c2.m_nCents);
}
```

Overload - operator

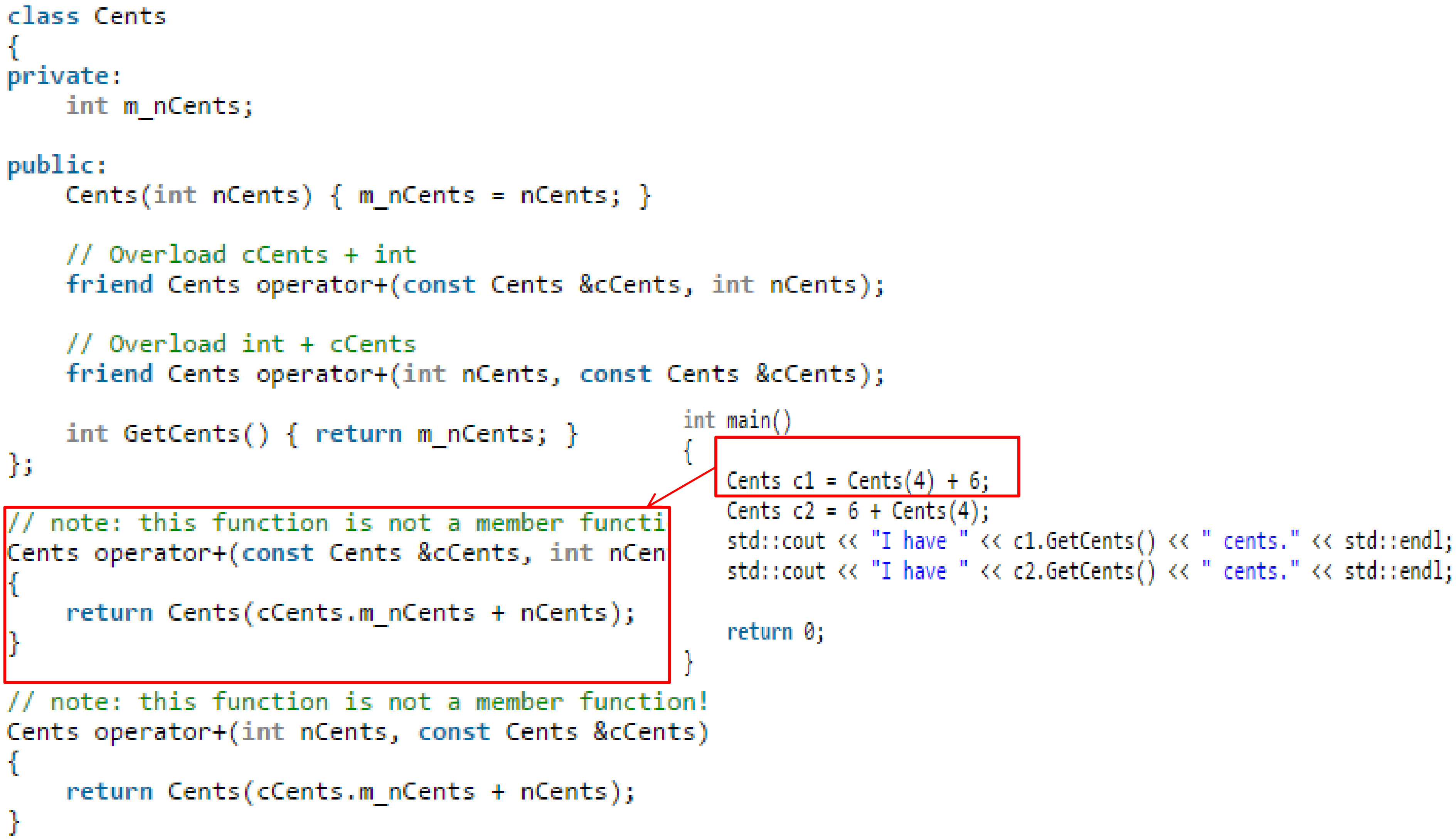


```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // overload Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    // overload Cents - Cents
    friend Cents operator-(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

// note: this function is not a member function!
Cents operator-(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator-(int, int)
    return Cents(c1.m_nCents - c2.m_nCents);
}
```

Overloading operators for operands of different types

- Cents(4) + 6 would call operator+(Cents, int).
- 6 + Cents(4) would call operator+(int, Cents).
- Consequently, whenever we overload binary operators for operands of different types, we actually need to write two functions — one for each case.

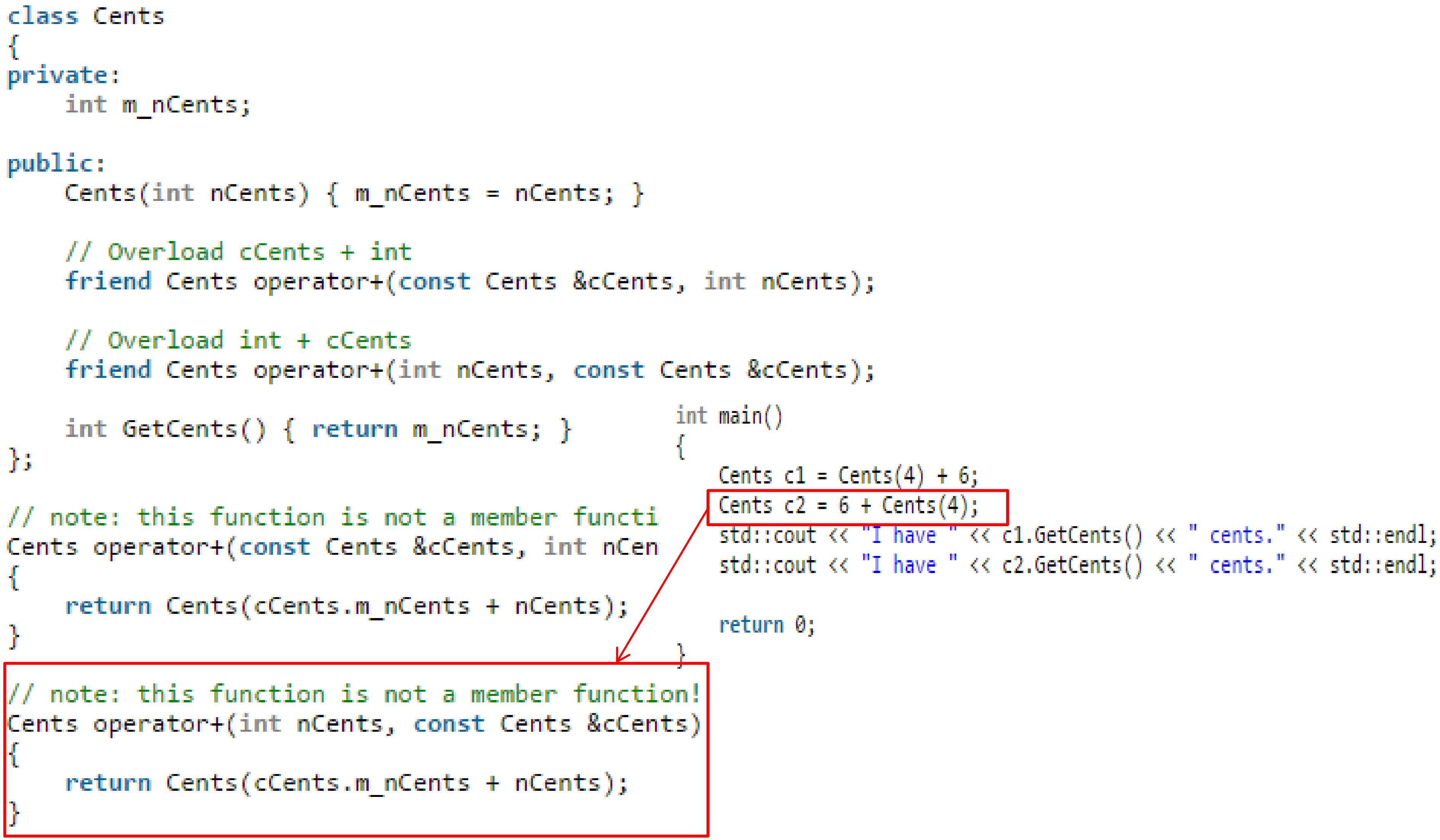
Overloading operators for operands of different types



```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    friend Cents operator+(const Cents &cCents, int nCents);

    // Overload int + cCents
    friend Cents operator+(int nCents, const Cents &cCents);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member functi
Cents operator+(const Cents &cCents, int nCen
{
    return Cents(cCents.m_nCents + nCents);
}

// note: this function is not a member function!
Cents operator+(int nCents, const Cents &cCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

```
int main()
{
    Cents c1 = Cents(4) + 6;
    Cents c2 = 6 + Cents(4);
    std::cout << "I have " << c1.GetCents() << " cents." << std::endl;
    std::cout << "I have " << c2.GetCents() << " cents." << std::endl;

    return 0;
}
```

Overloading operators for operands of different types



```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    friend Cents operator+(const Cents &cCents, int nCents);

    // Overload int + cCents
    friend Cents operator+(int nCents, const Cents &cCents);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member functi
Cents operator+(const Cents &cCents, int nCen
{
    return Cents(cCents.m_nCents + nCents);
}

// note: this function is not a member function!
Cents operator+(int nCents, const Cents &cCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

```
int main()
{
    Cents c1 = Cents(4) + 6;
    Cents c2 = 6 + Cents(4);
    std::cout << "I have " << c1.GetCents() << " cents." << std::endl;
    std::cout << "I have " << c2.GetCents() << " cents." << std::endl;

    return 0;
}
```

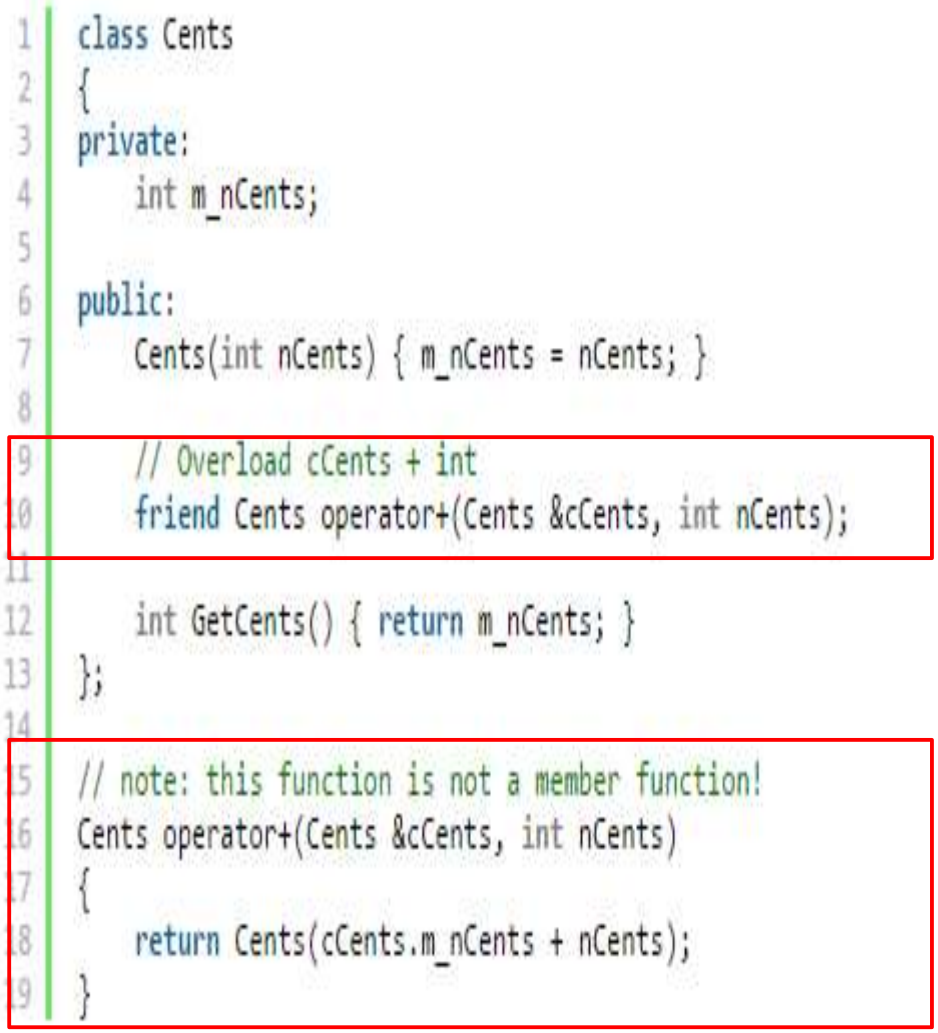
Overloading operators using member functions

- when the operator does not modify it’s operands, it’s best to implement the overloaded operator as a friend function of the class.
- For operators that do modify their operands, we typically overload the operator using a member function of the class.

Overloading operators using member functions

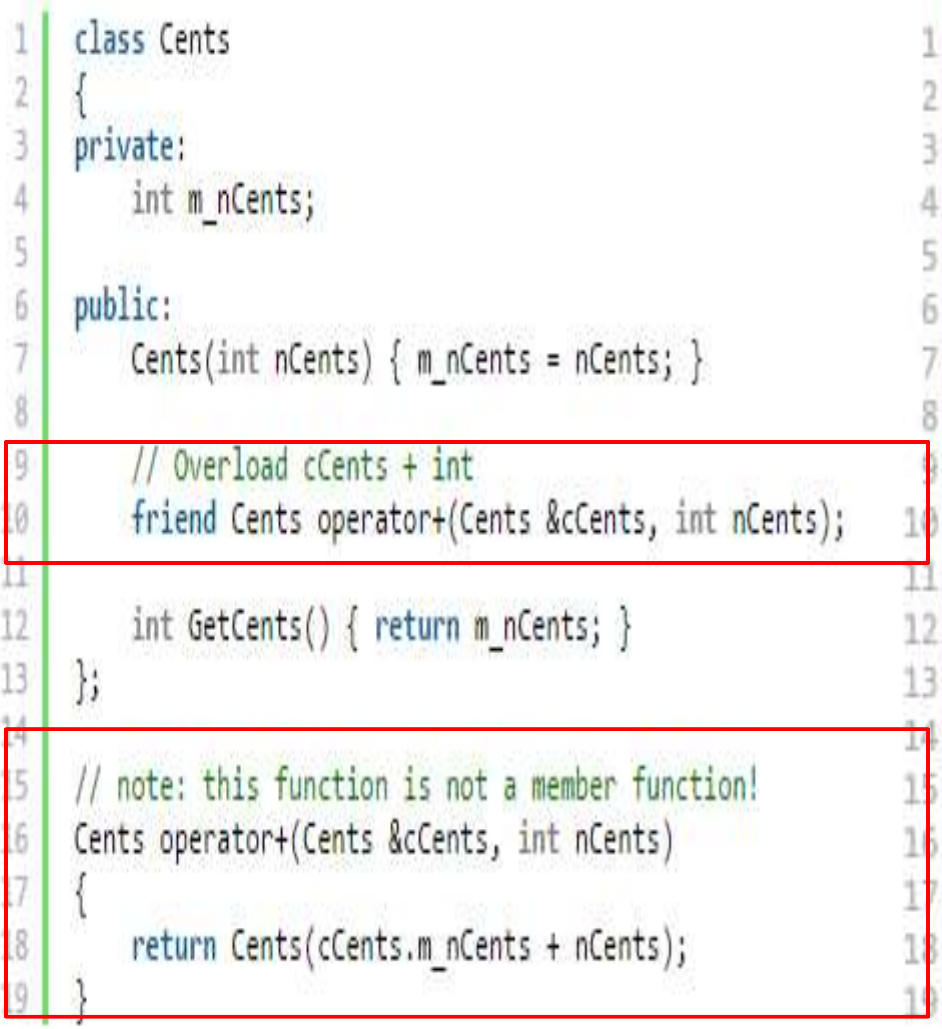
- Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:
  - The leftmost operand of the overloaded operator must be an object of the class type.
  - The leftmost operand becomes the implicit ***this** parameter. All other operands become function parameters.

Overloading operators using member functions

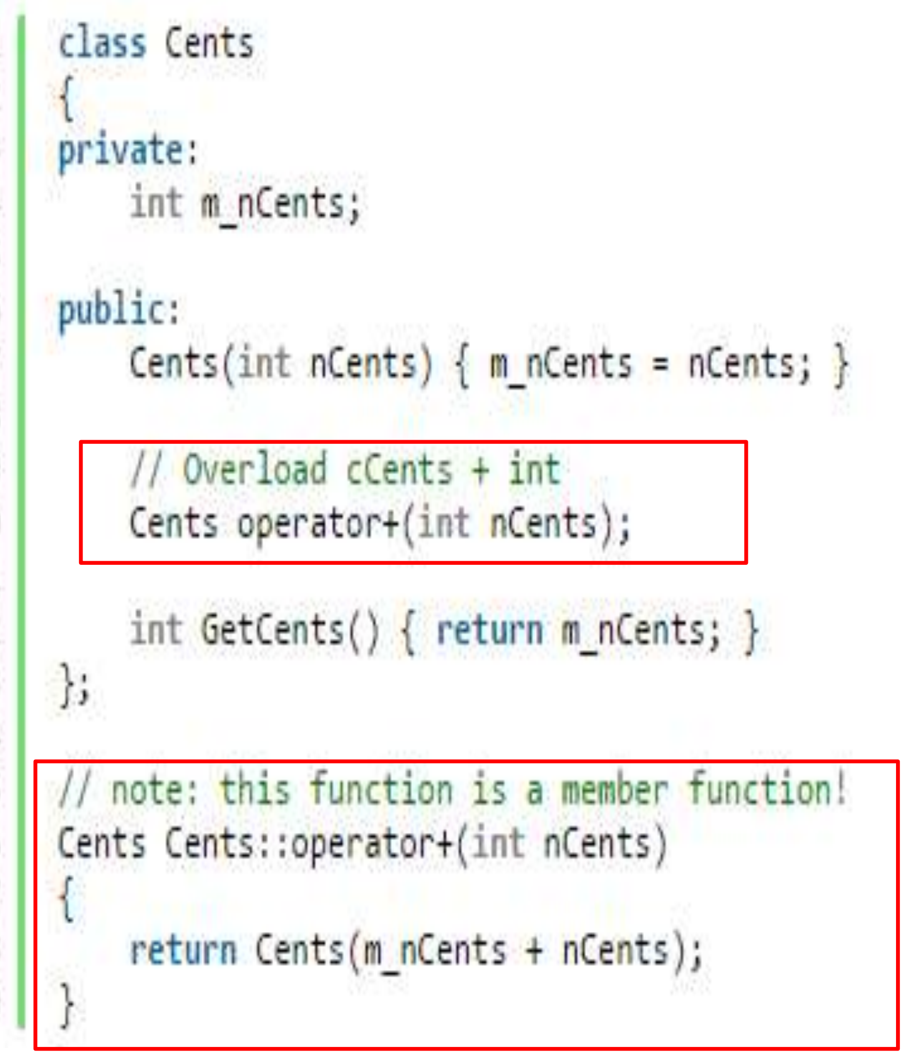


```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    friend Cents operator+(Cents &cCents, int nCents);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

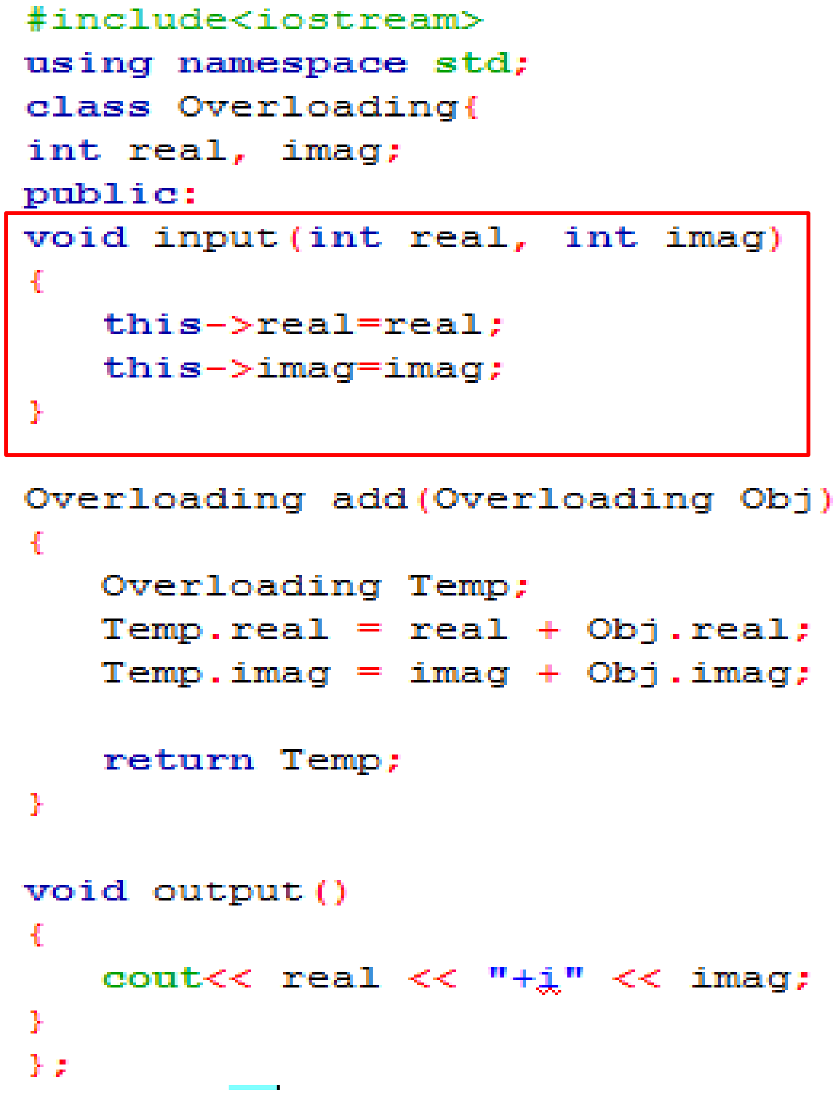
Overloading operators using member functions



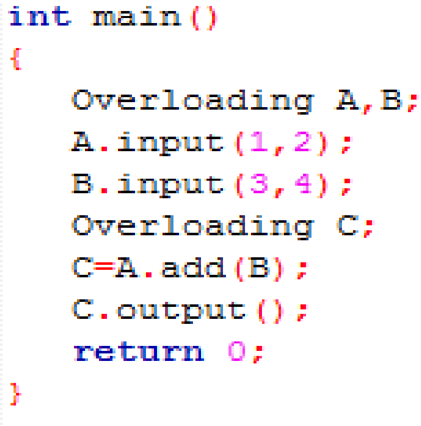
```cpp
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    friend Cents operator+(Cents &cCents, int nCents);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```



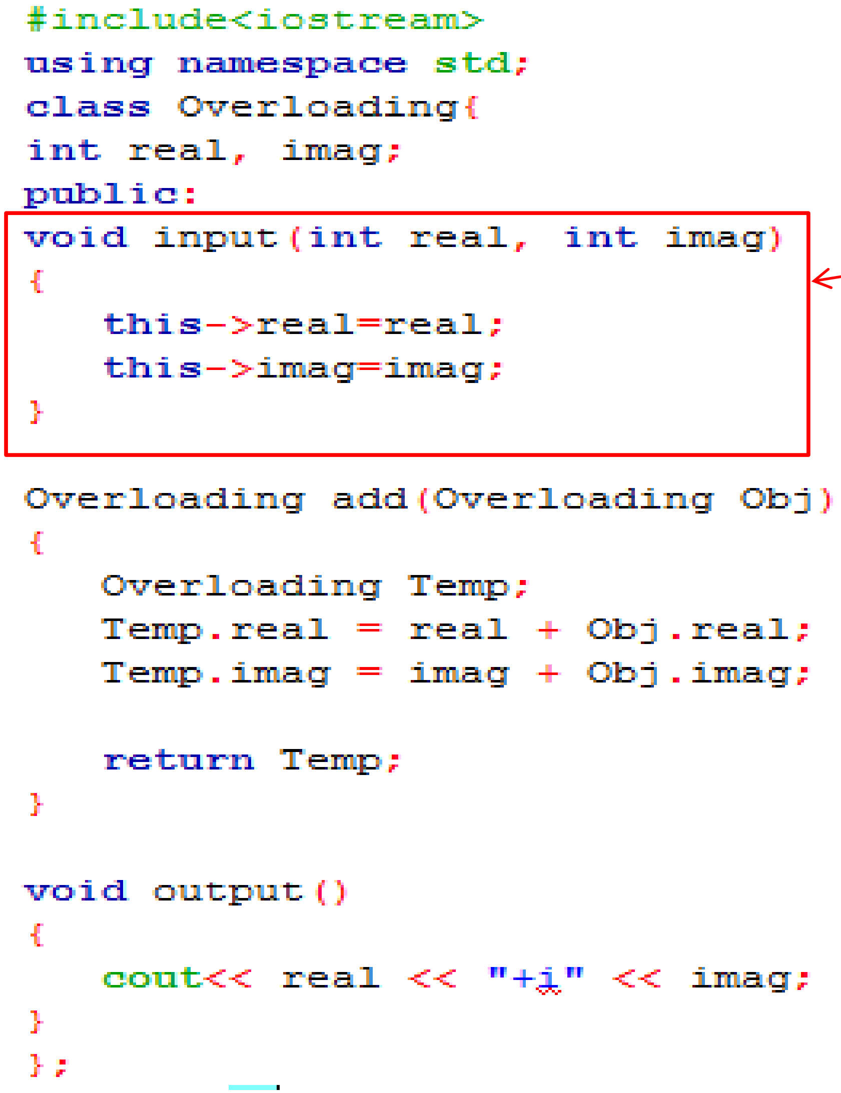
```cpp
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    Cents operator+(int nCents);

    int GetCents() { return m_nCents; }
};

// note: this function is a member function!
Cents Cents::operator+(int nCents)
{
    return Cents(m_nCents + nCents);
}
```



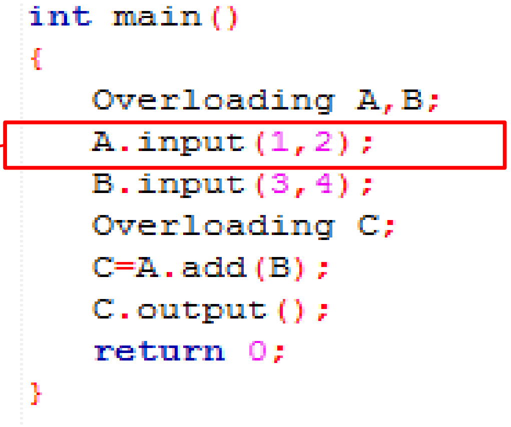
```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```



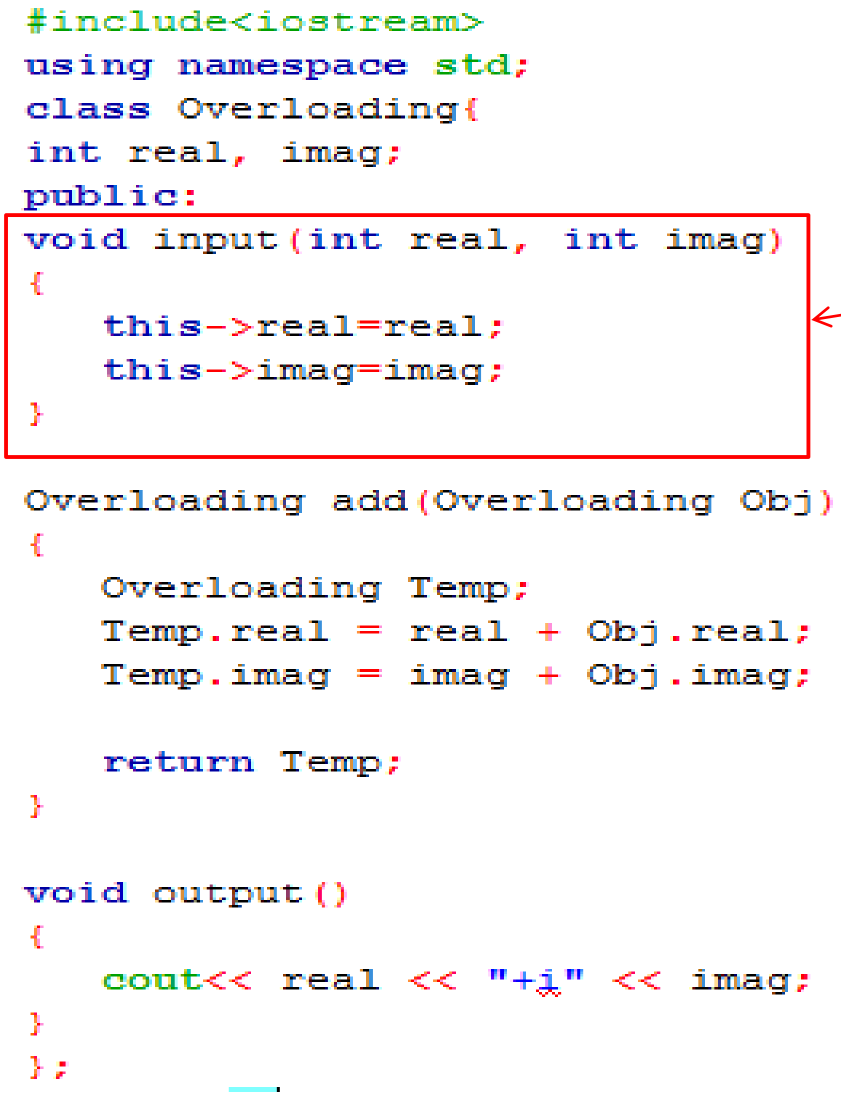
```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```



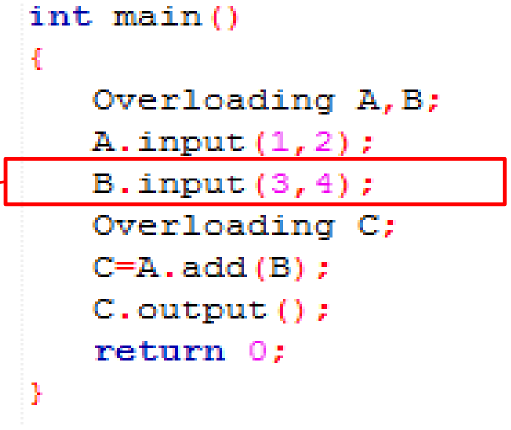
```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```



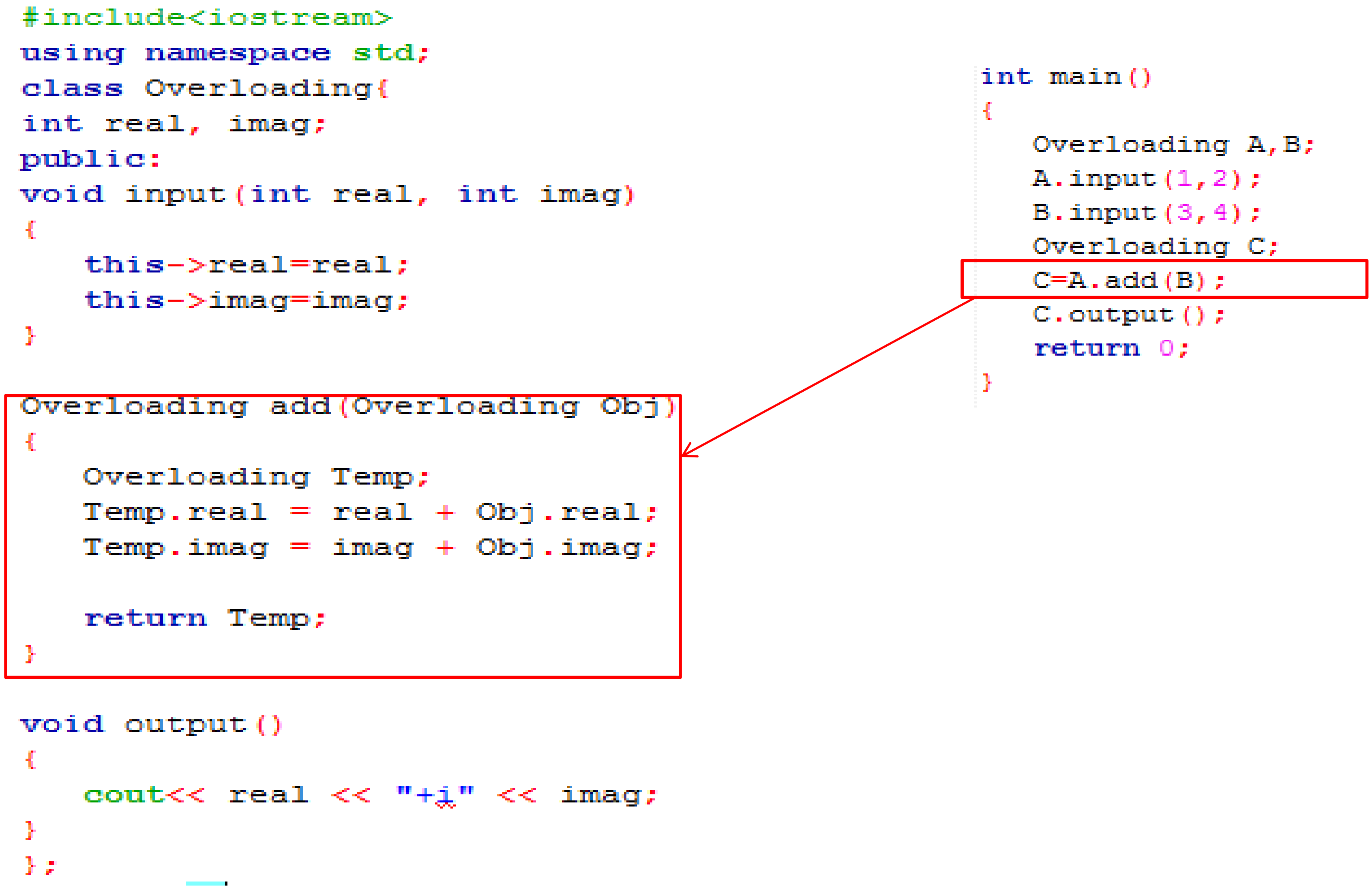
```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

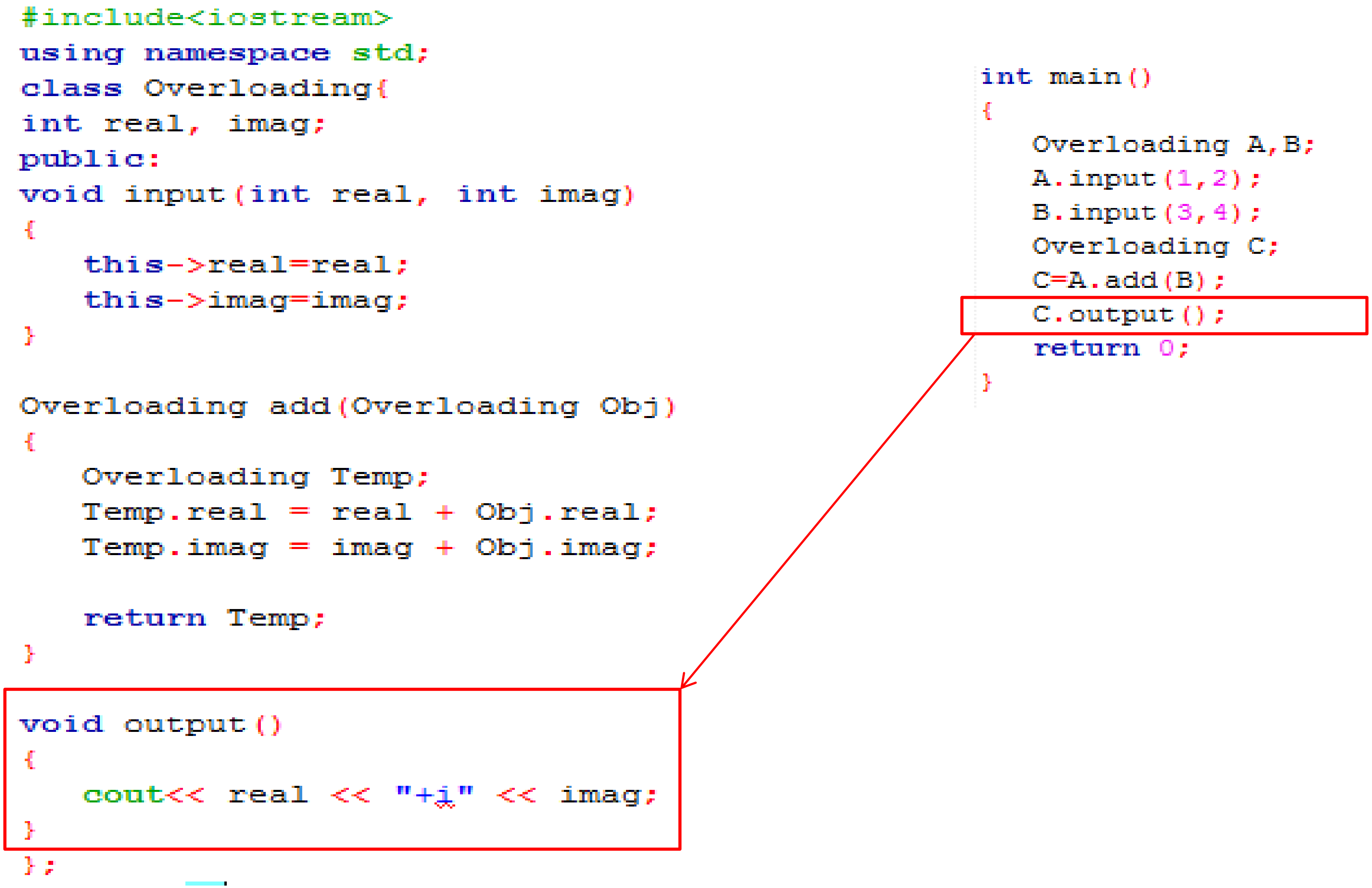


```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```



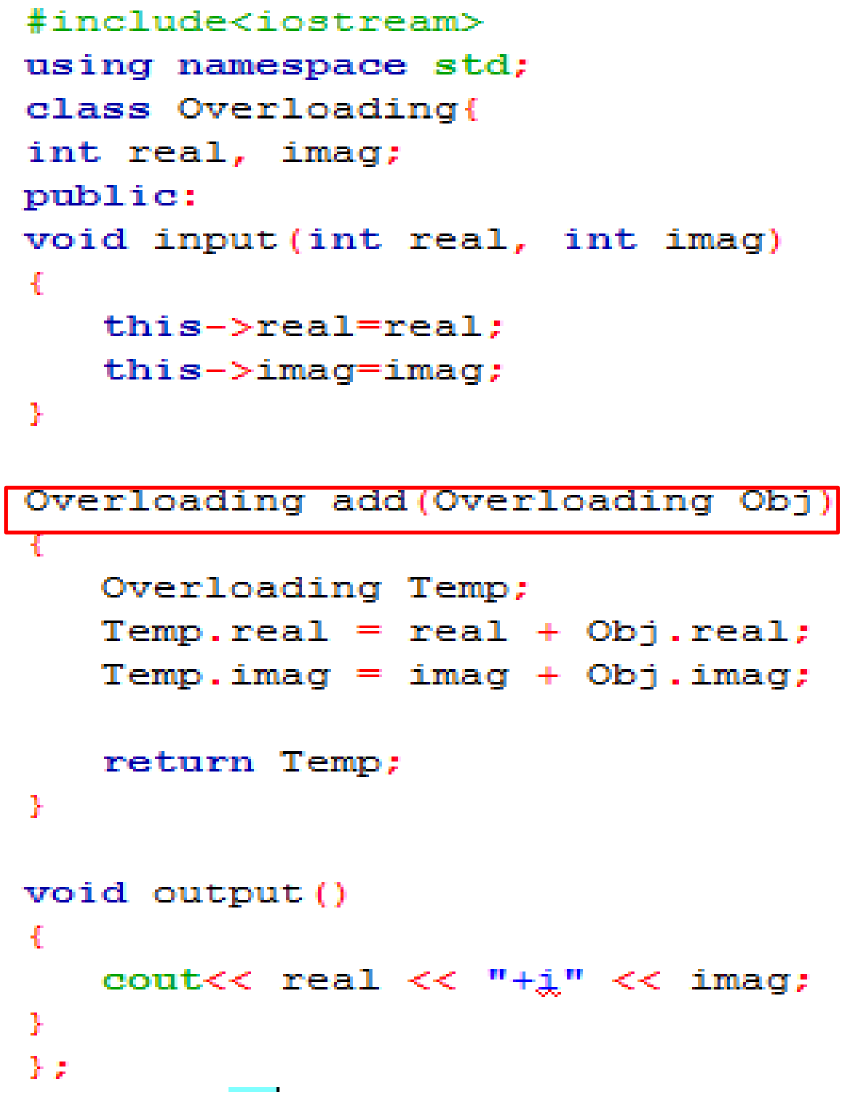
```cpp
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

```cpp
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

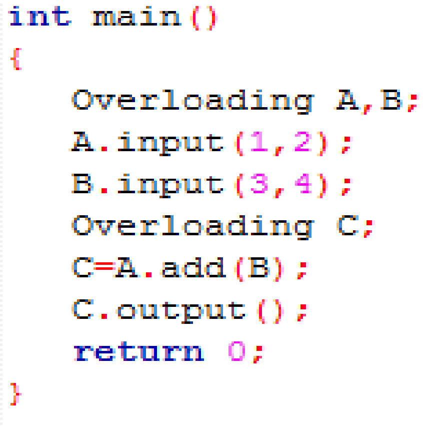


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

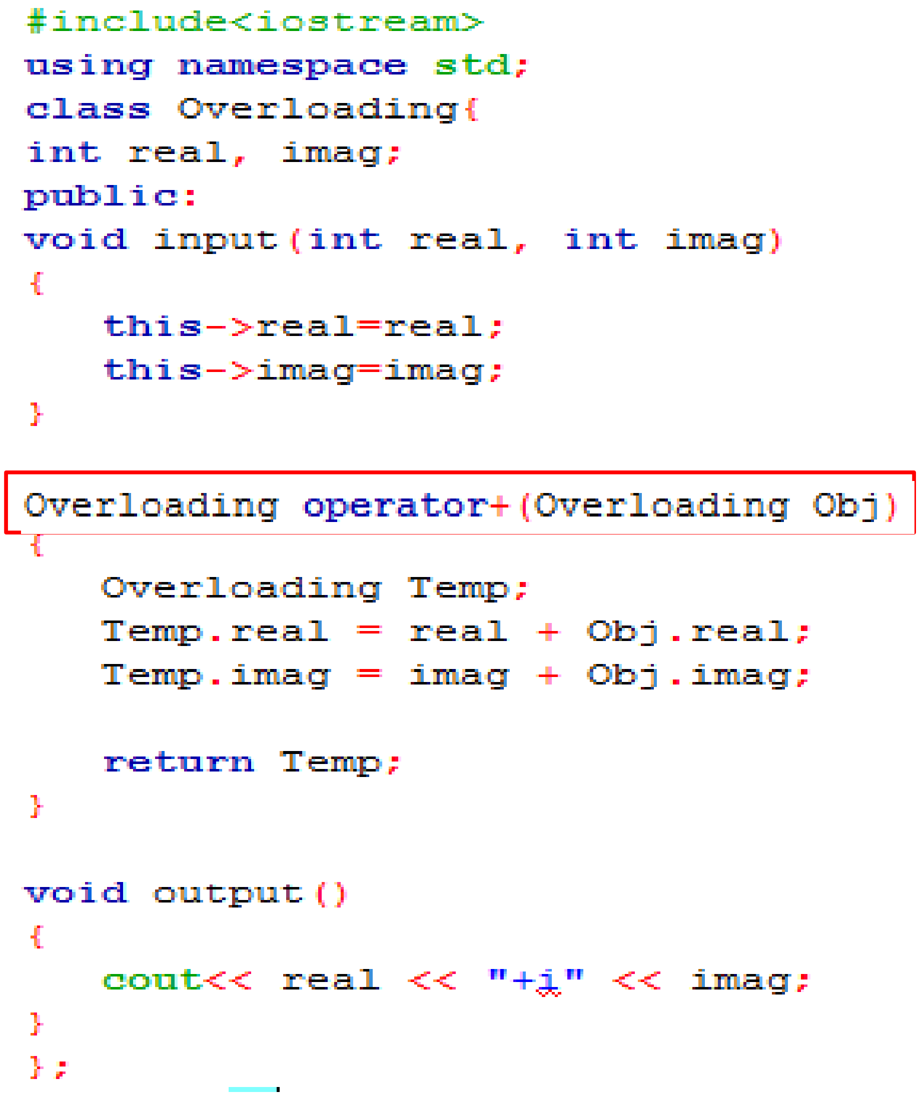
```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```



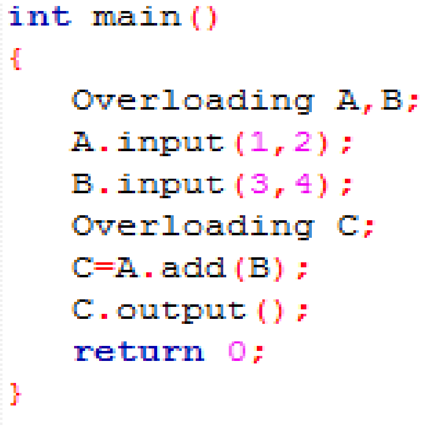
```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```



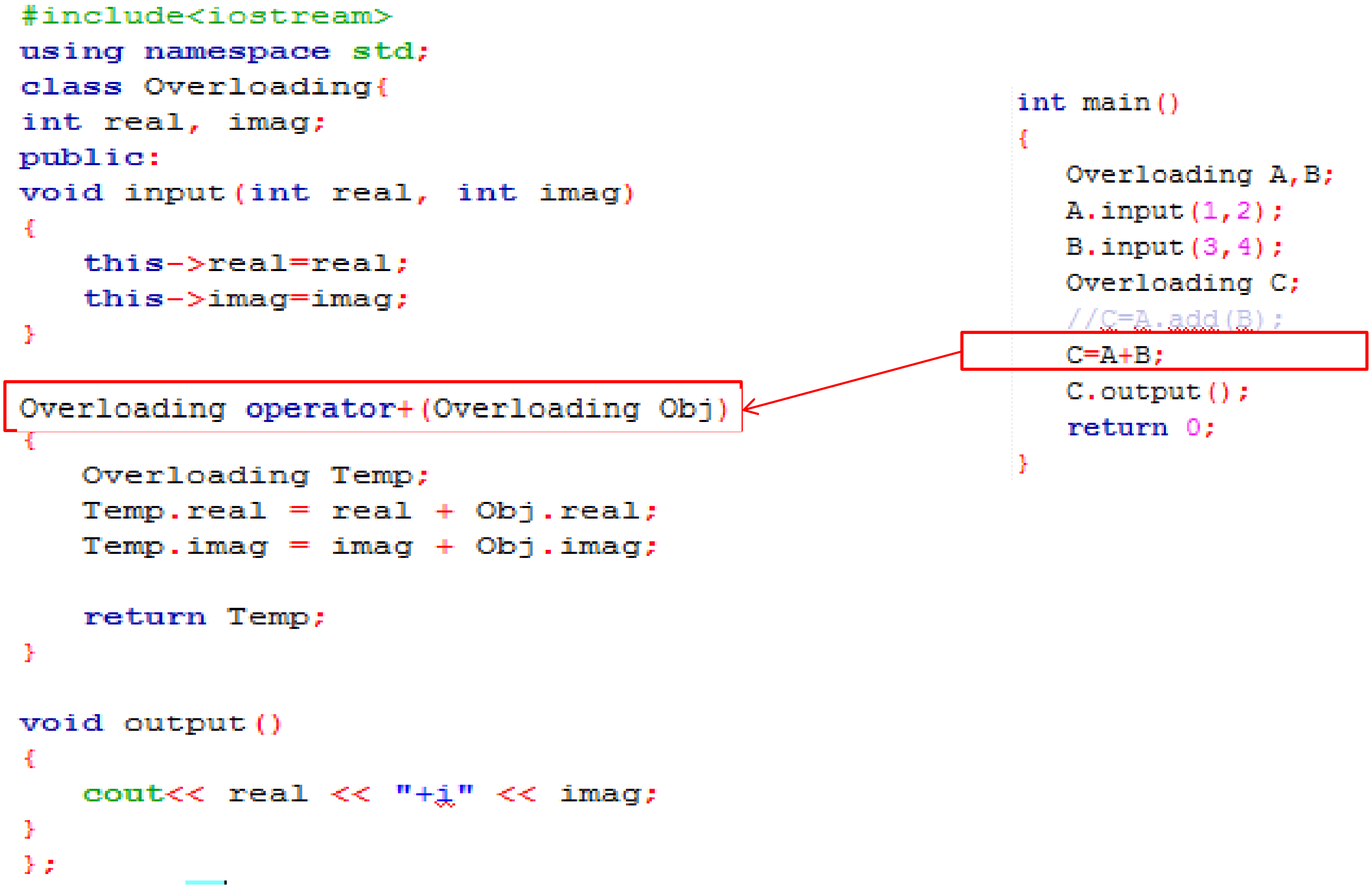
```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

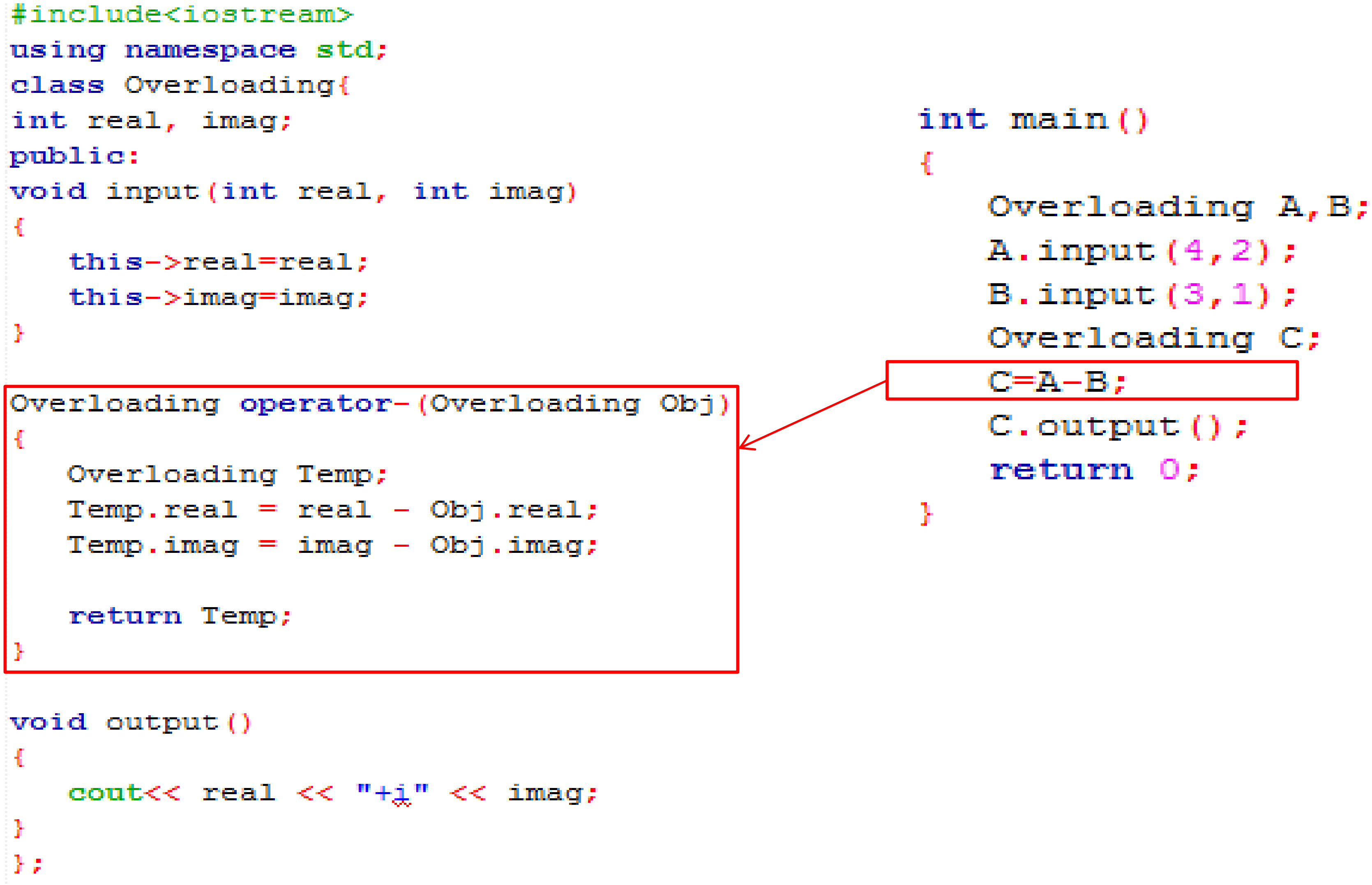


```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```



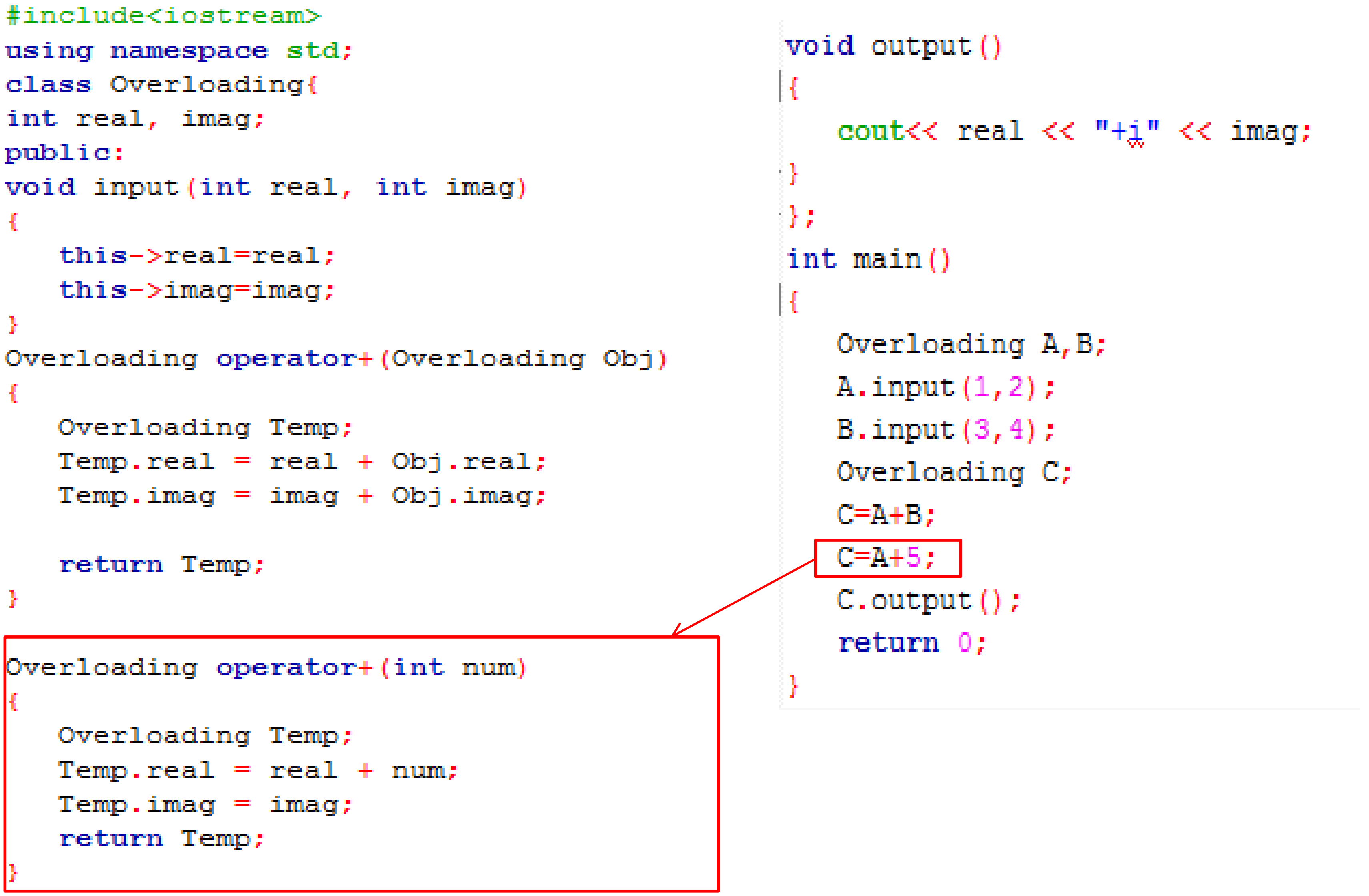
```cpp
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

```cpp
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    //C=A.add(B);
    C=A+B;
    C.output();
    return 0;
}
```

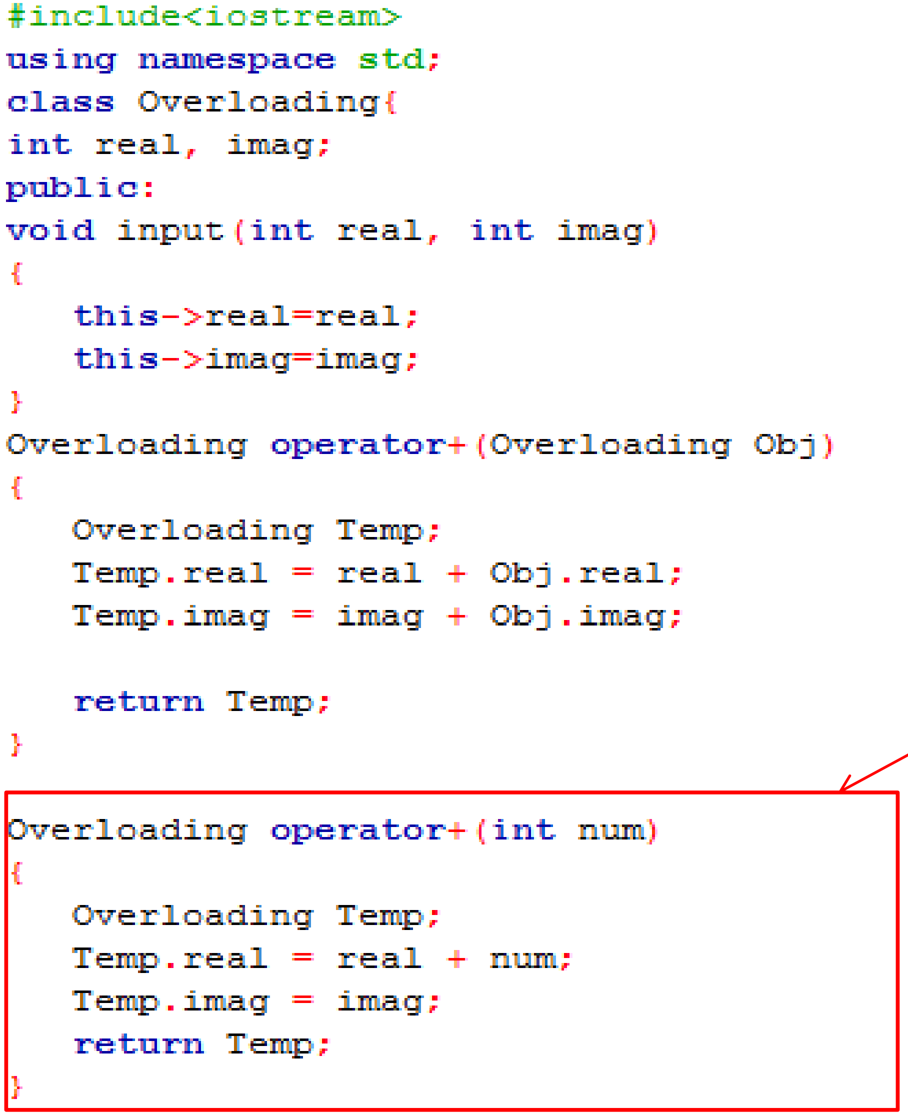


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator-(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real - Obj.real;
    Temp.imag = imag - Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

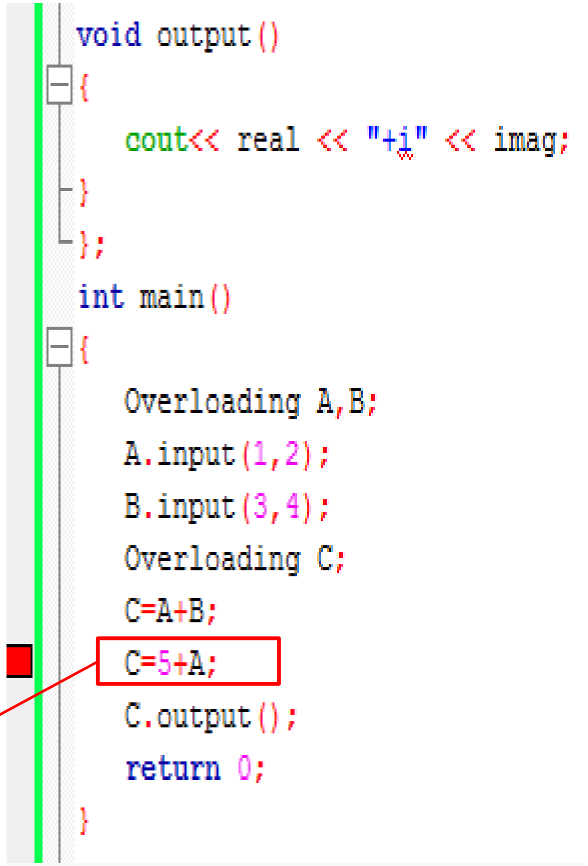
```
int main()
{
    Overloading A,B;
    A.input(4,2);
    B.input(3,1);
    Overloading C;
    C=A-B;
    C.output();
    return 0;
}
```



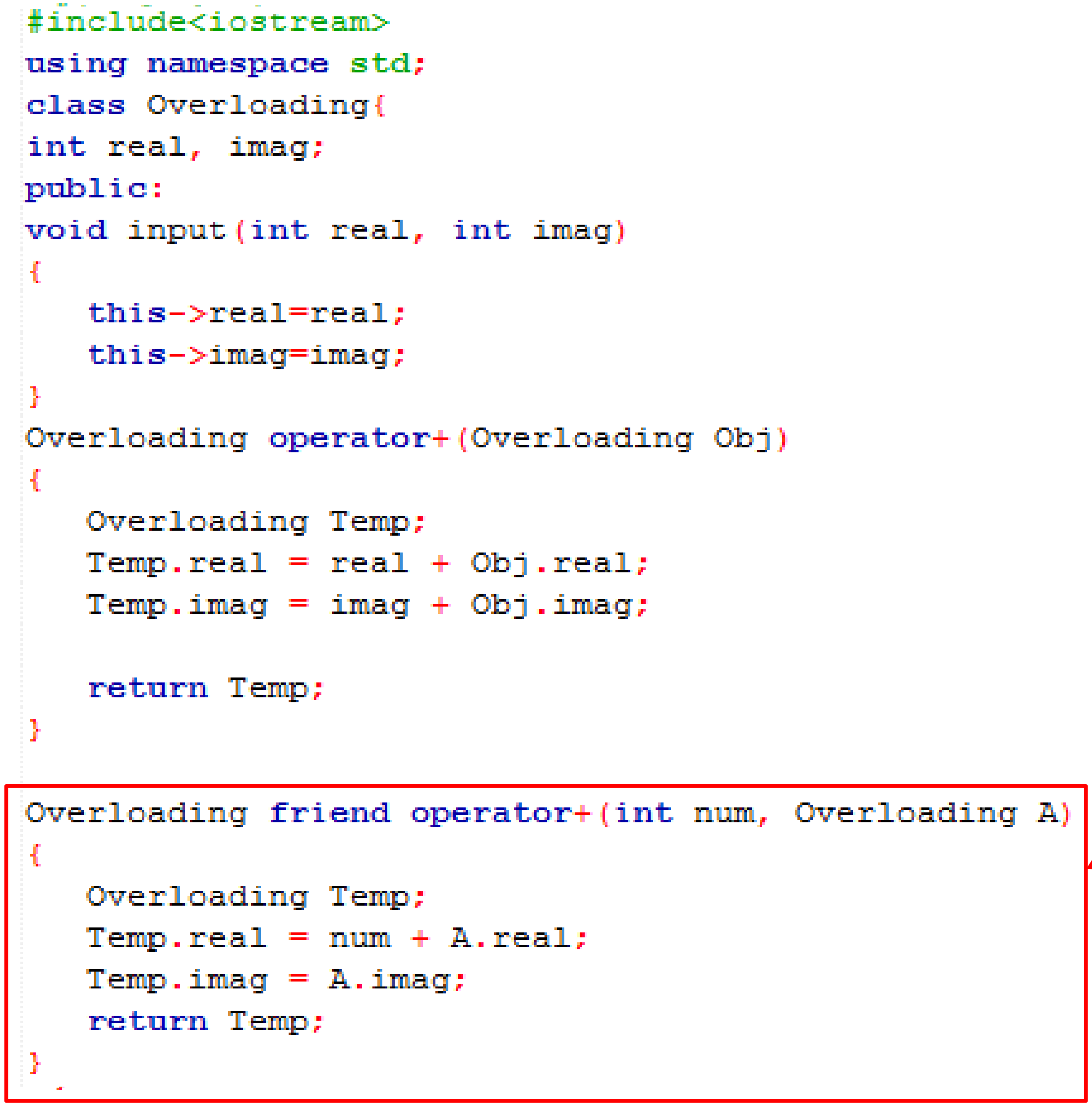
```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}
Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}
Overloading operator+(int num)
{
    Overloading Temp;
    Temp.real = real + num;
    Temp.imag = imag;
    return Temp;
}
void output()
{
    cout<< real << "+i" << imag;
}
};
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A+B;
    C=A+5;
    C.output();
    return 0;
}
```



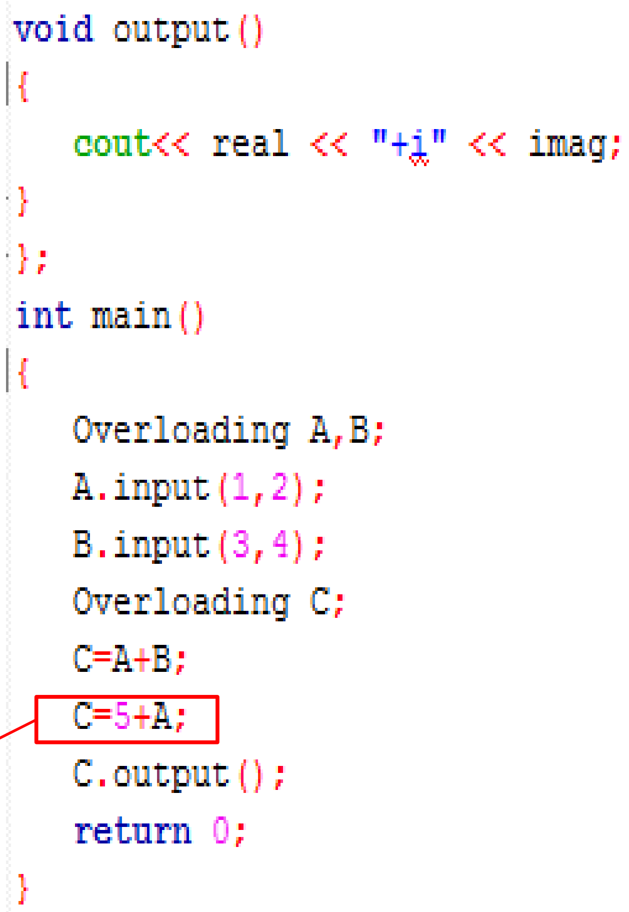
```cpp
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}
Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

Overloading operator+(int num)
{
    Overloading Temp;
    Temp.real = real + num;
    Temp.imag = imag;
    return Temp;
}
```



```cpp
void output()
{
    cout<< real << "+i" << imag;
}
};
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A+B;
    C=5+A;
    C.output();
    return 0;
}
```



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}
Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

Overloading friend operator+(int num, Overloading A)
{
    Overloading Temp;
    Temp.real = num + A.real;
    Temp.imag = A.imag;
    return Temp;
}
```



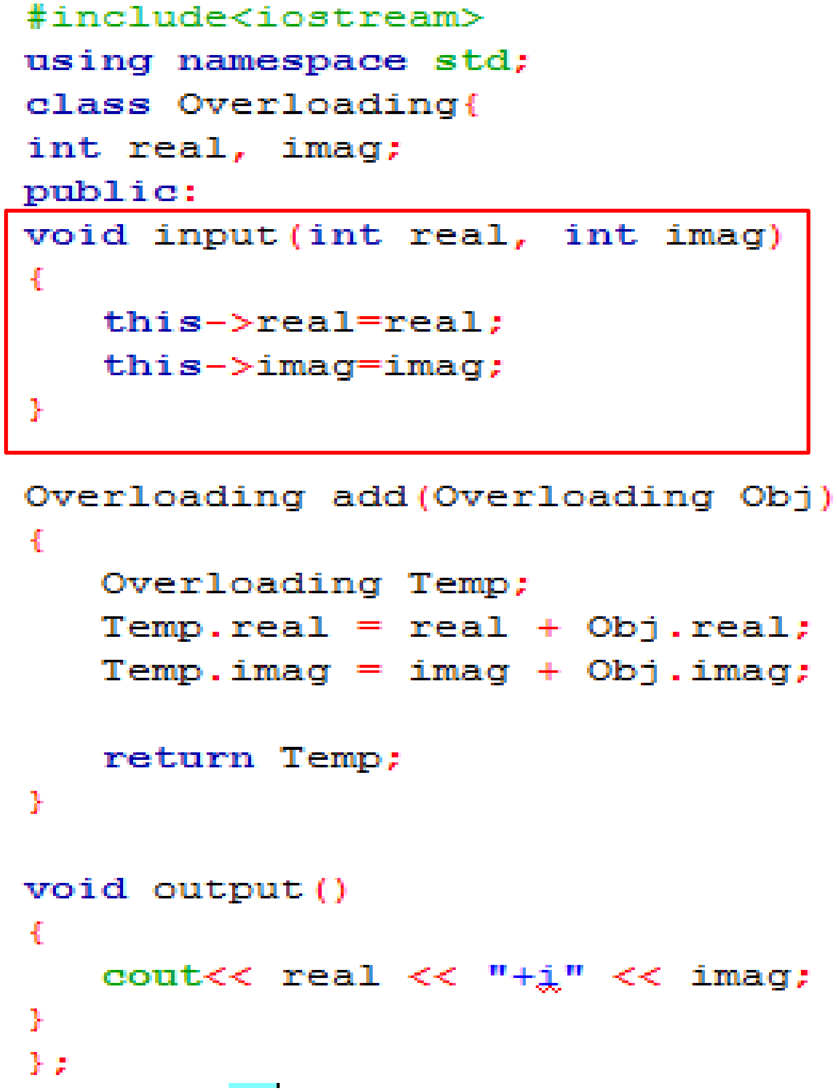
```
void output()
{
    cout<< real << "+i" << imag;
}
};
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A+B;
    C=5+A;
    C.output();
    return 0;
}
```

References

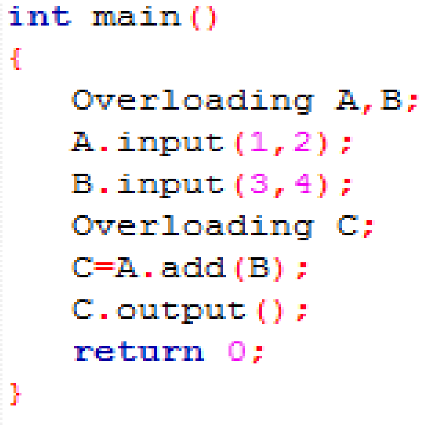
- http://www.learncpp.com
- http://www.cplusplus.com/articles/ENywvCM9/

OPERATOR OVERLOADING

A program for addition

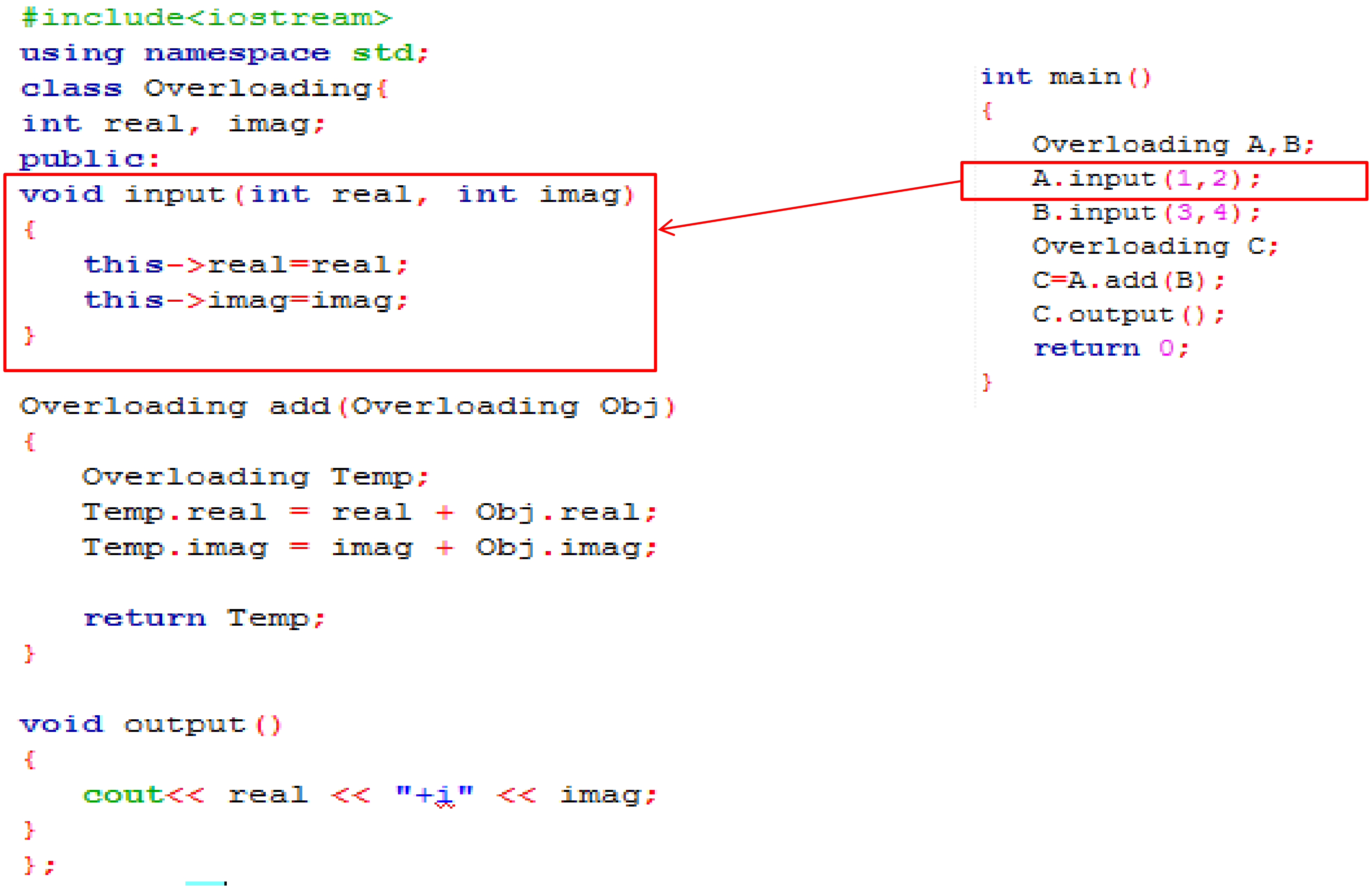


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```



```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

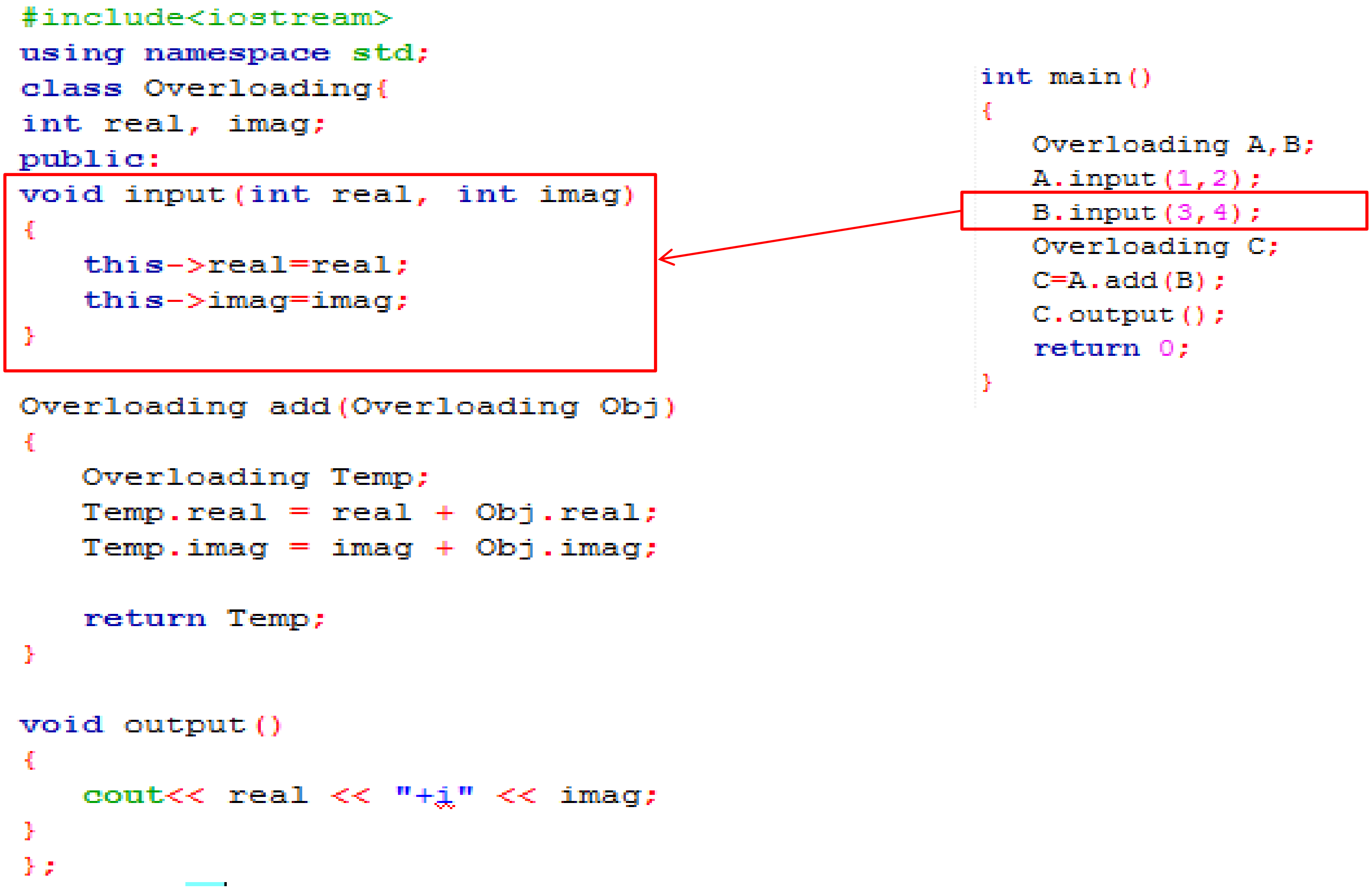
A program for addition



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

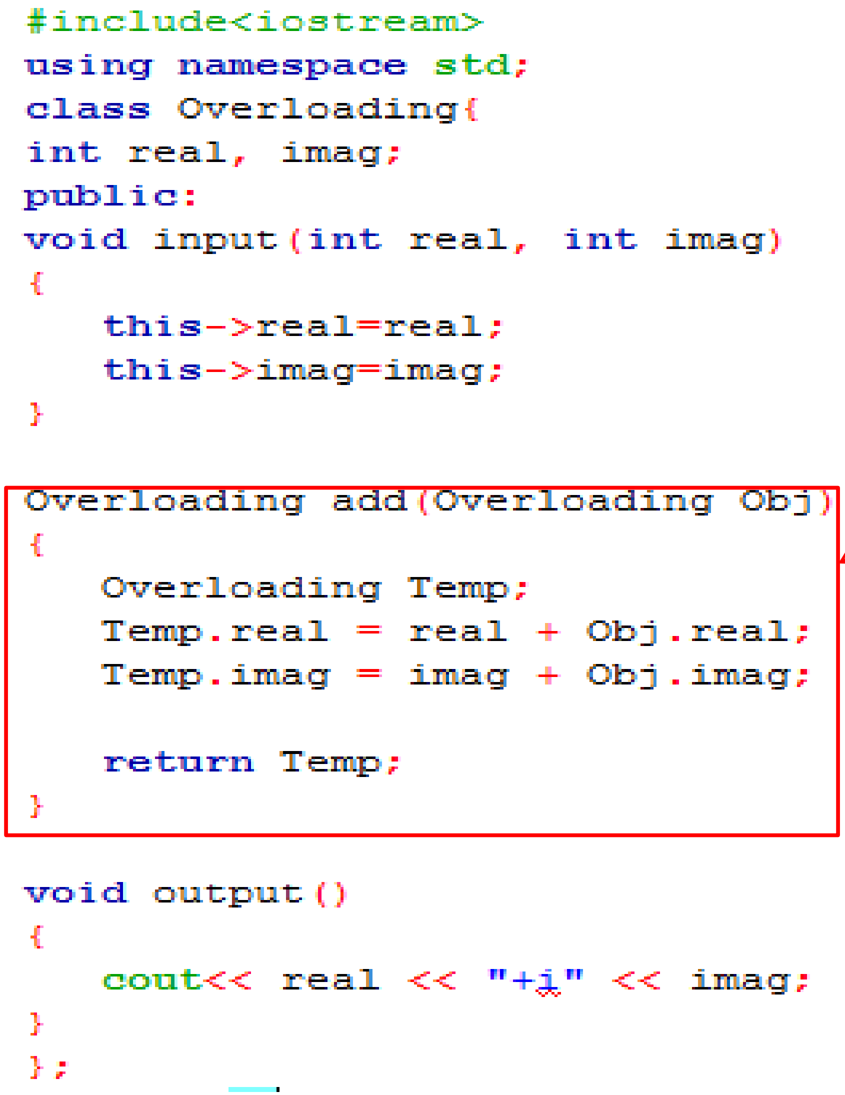
A program for addition



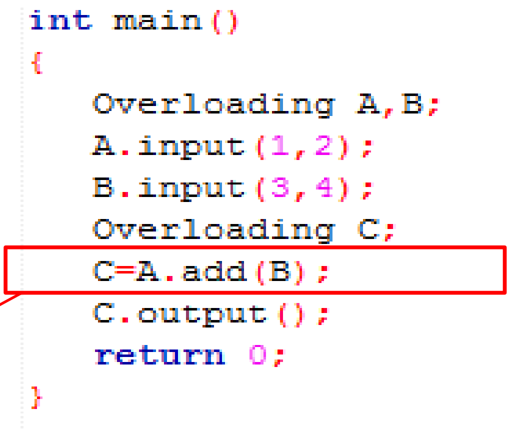
```cpp
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

```cpp
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

A program for addition

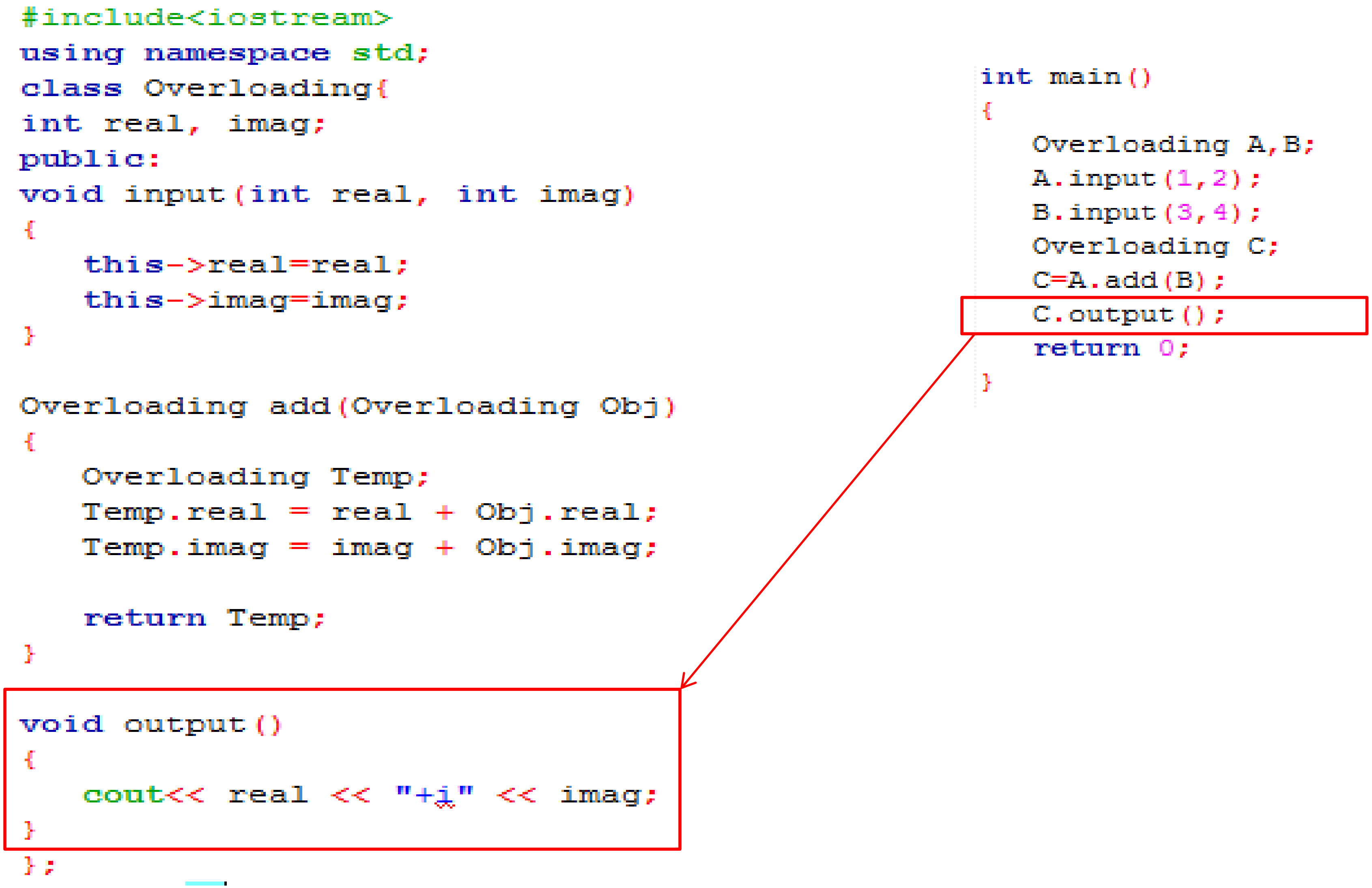


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```



```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

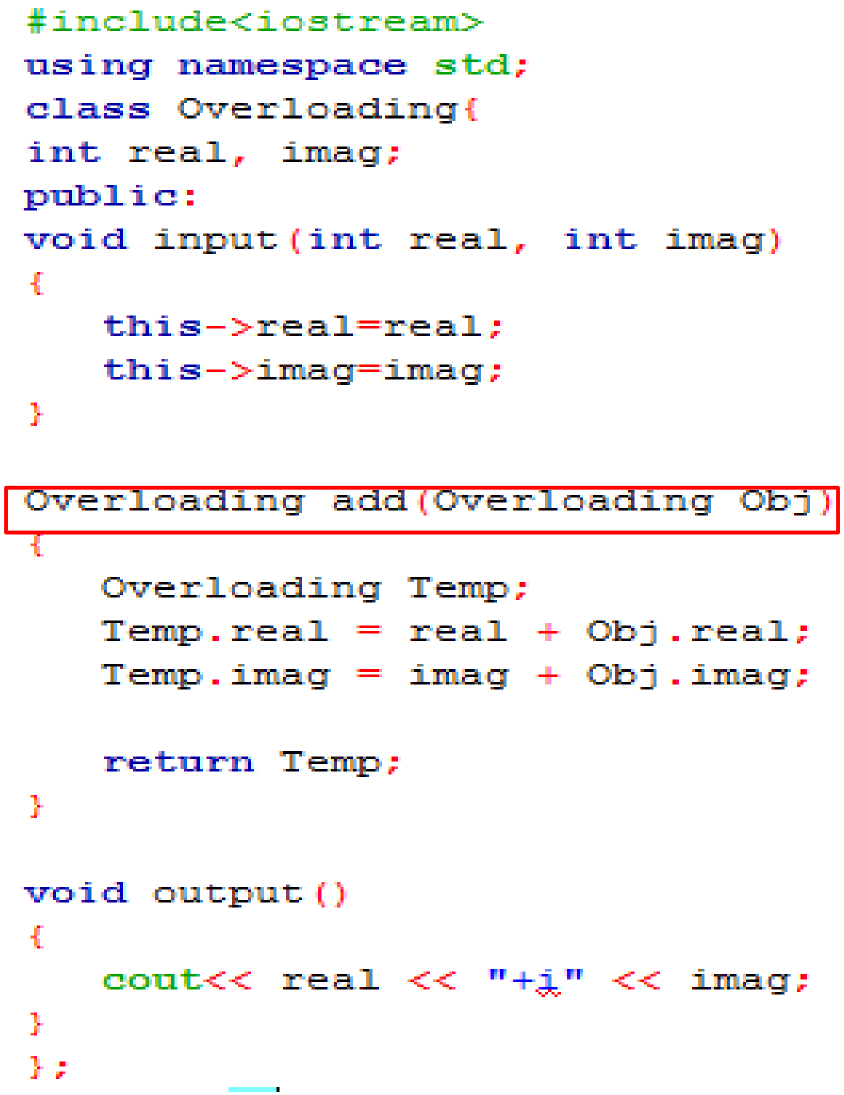
A program for addition



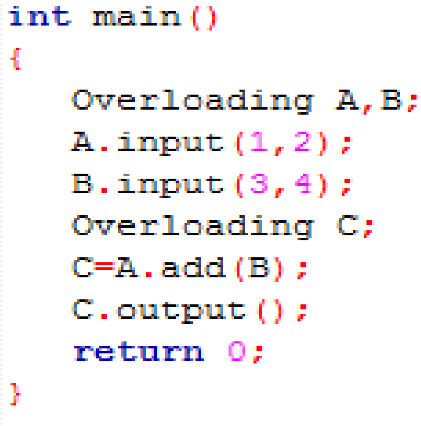
```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

A program for addition

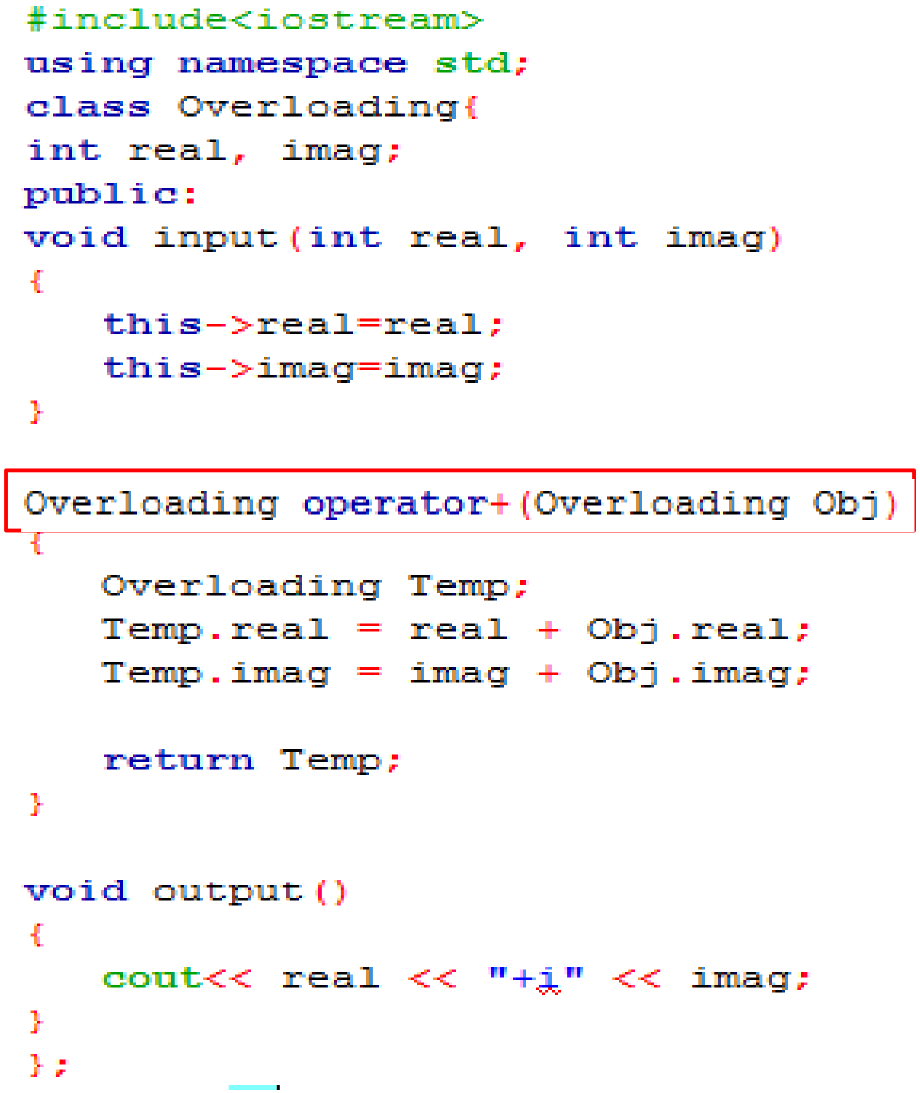


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading add(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

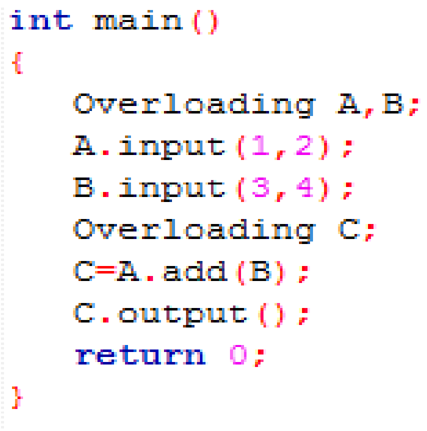


```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

Binary + Operator overloading

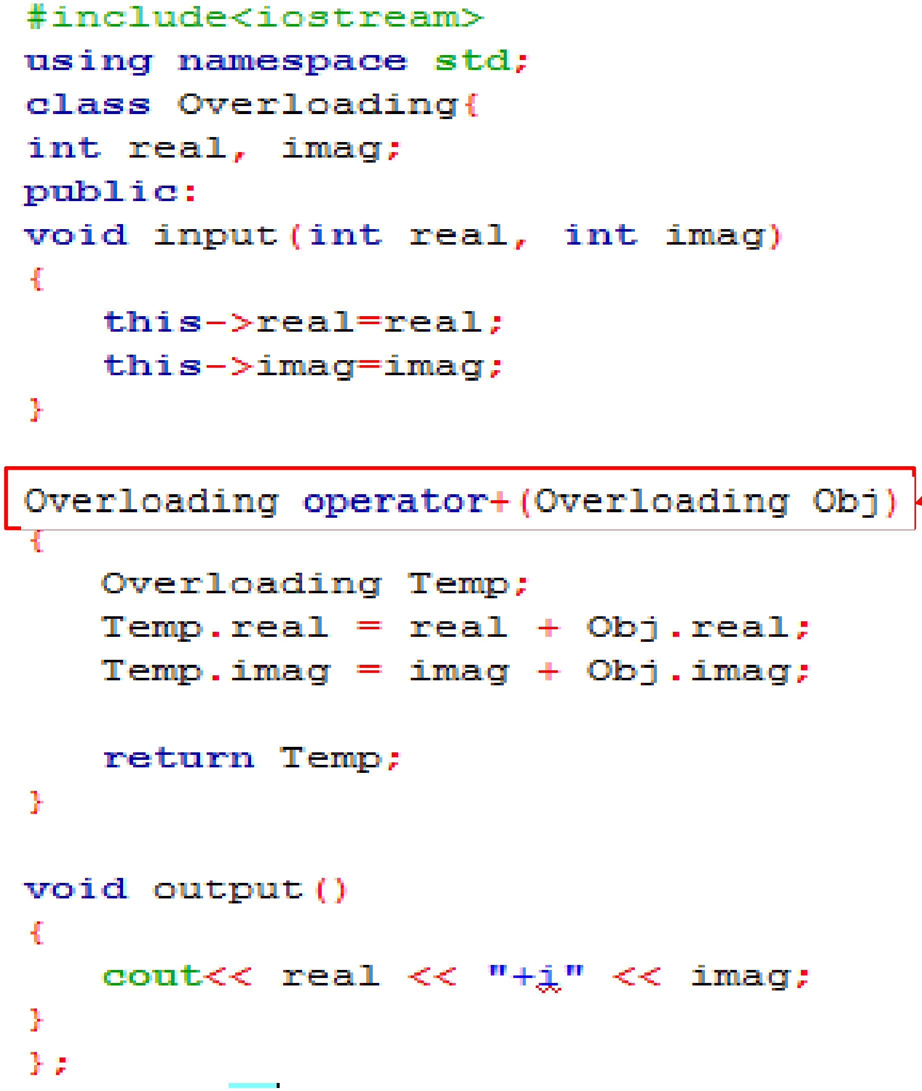


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

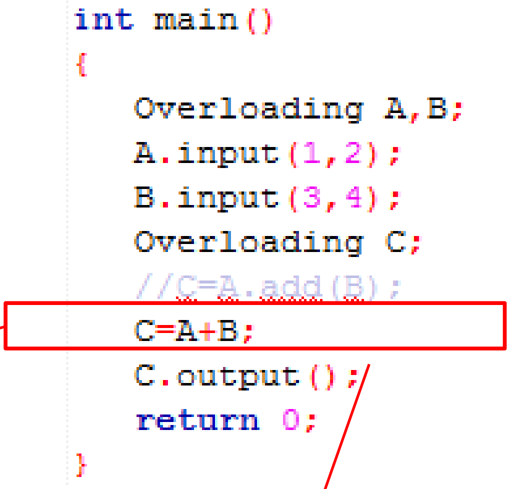


```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A.add(B);
    C.output();
    return 0;
}
```

Binary + Operator overloading



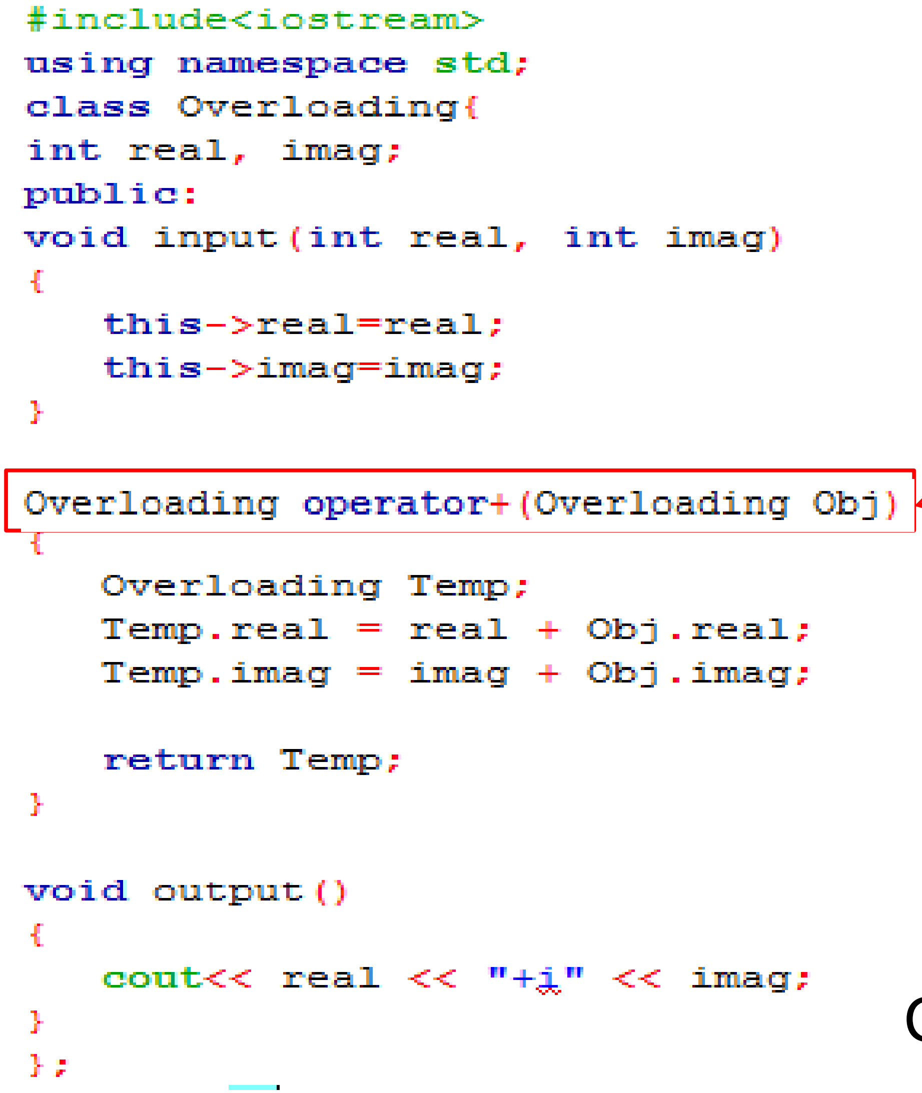
```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```



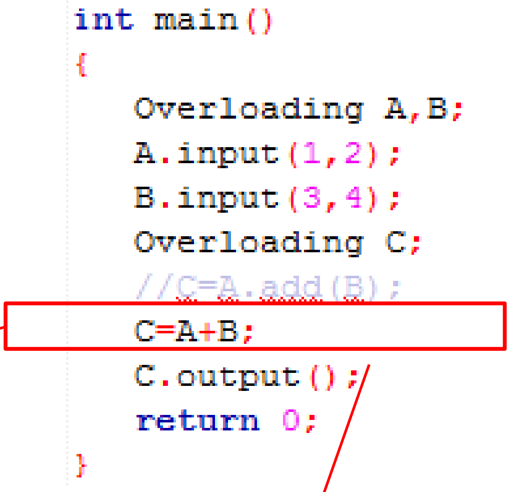
```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    //C=A.add(B);
    C=A+B;
    C.output();
    return 0;
}
```

C=A.operator+(B)

Binary + Operator overloading



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

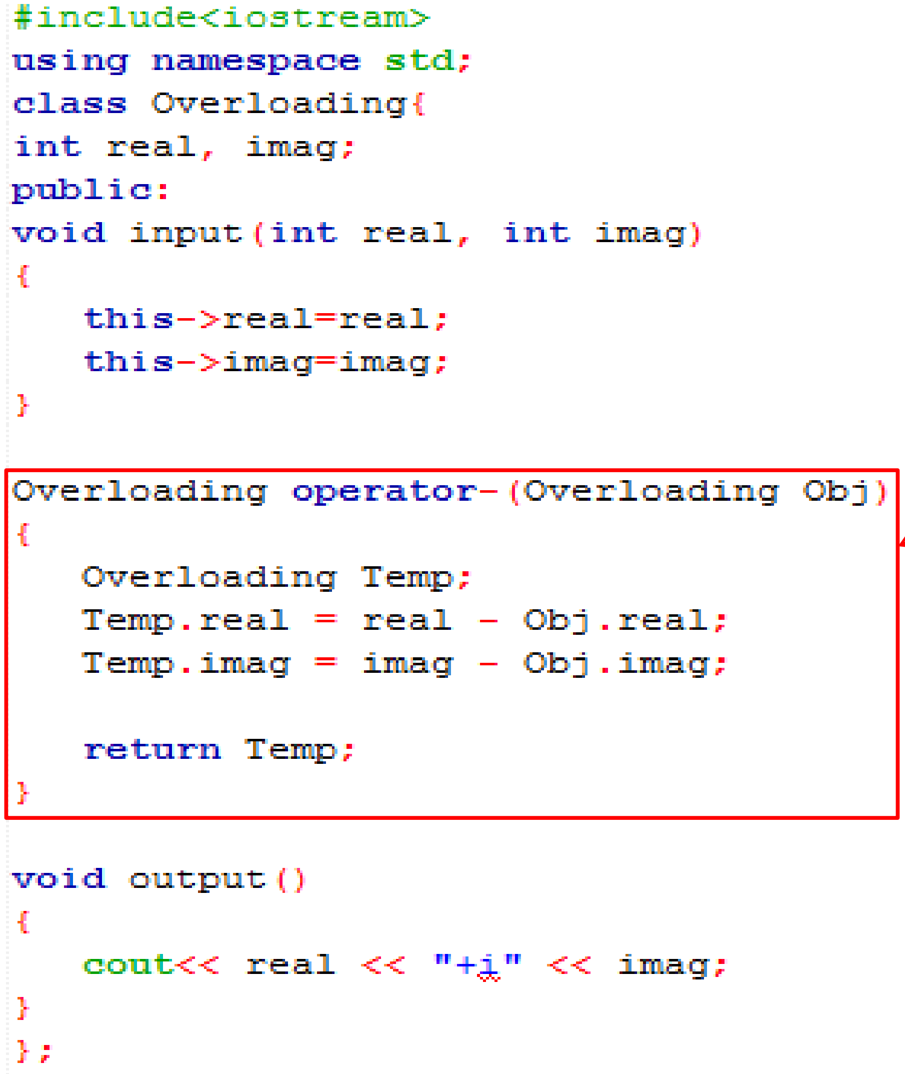


```
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    //C=A.add(B);
    C=A+B;
    C.output();
    return 0;
}
```

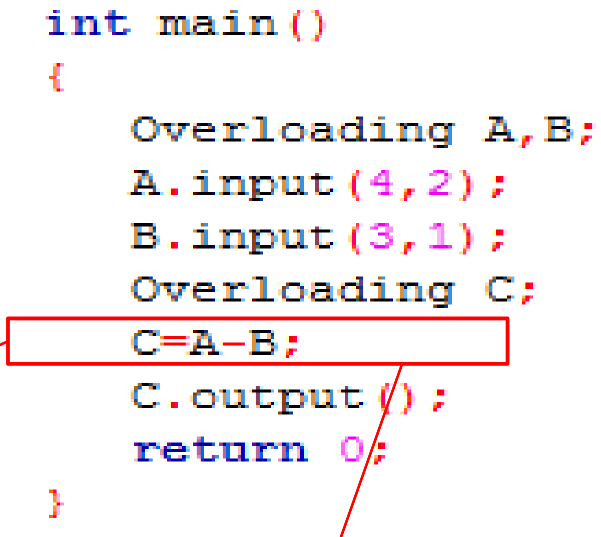
C=A.operator+(B)

Can you do it using friend function?

Binary - Operator overloading



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}

Overloading operator-(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real - Obj.real;
    Temp.imag = imag - Obj.imag;

    return Temp;
}

void output()
{
    cout<< real << "+i" << imag;
}
};
```

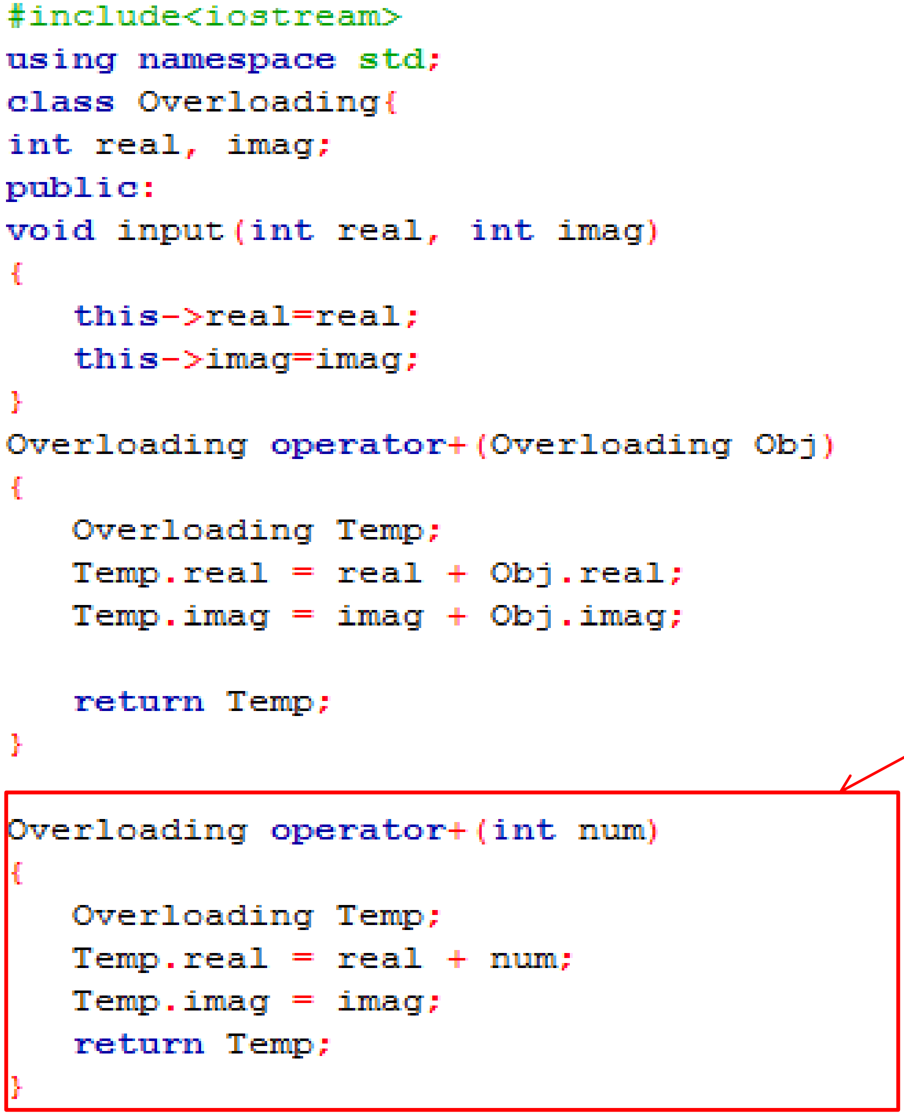


```
int main()
{
    Overloading A,B;
    A.input(4,2);
    B.input(3,1);
    Overloading C;
    C=A-B;
    C.output();
    return 0;
}
```

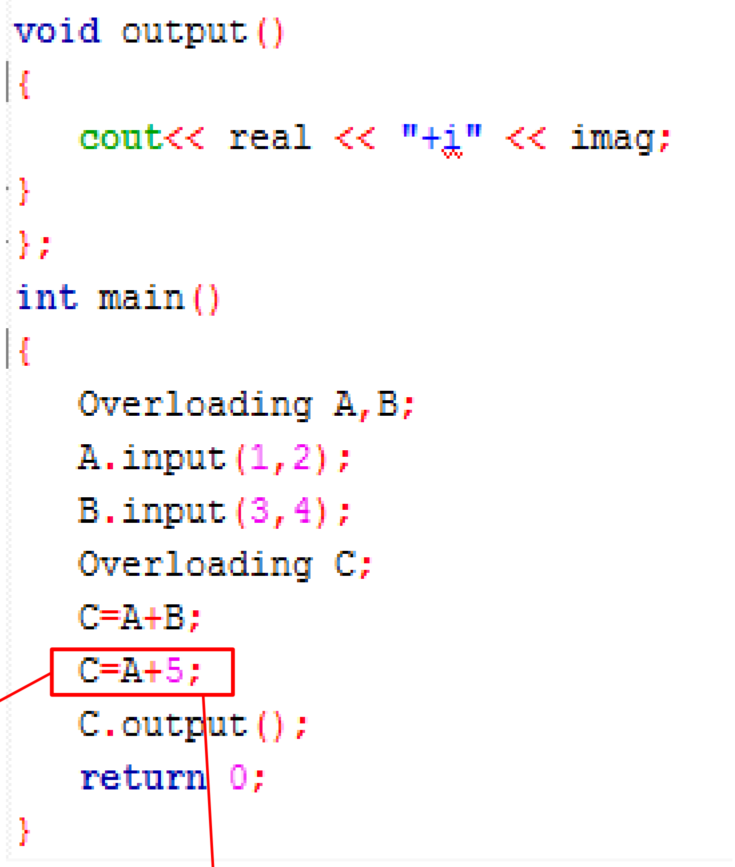
C=A.operator-(B)

Try to overload *,/,%

Overloading operators for operands of different types



```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}
Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

Overloading operator+(int num)
{
    Overloading Temp;
    Temp.real = real + num;
    Temp.imag = imag;
    return Temp;
}
```

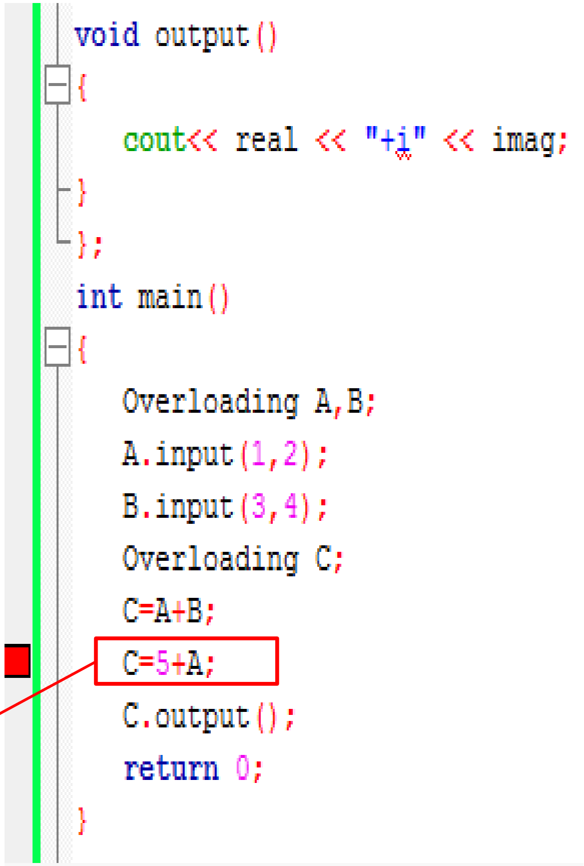


```
void output()
{
    cout<< real << "+i" << imag;
}
};
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A+B;
    C=A+5;
    C.output();
    return 0;
}
```

C=A.operator+(5)

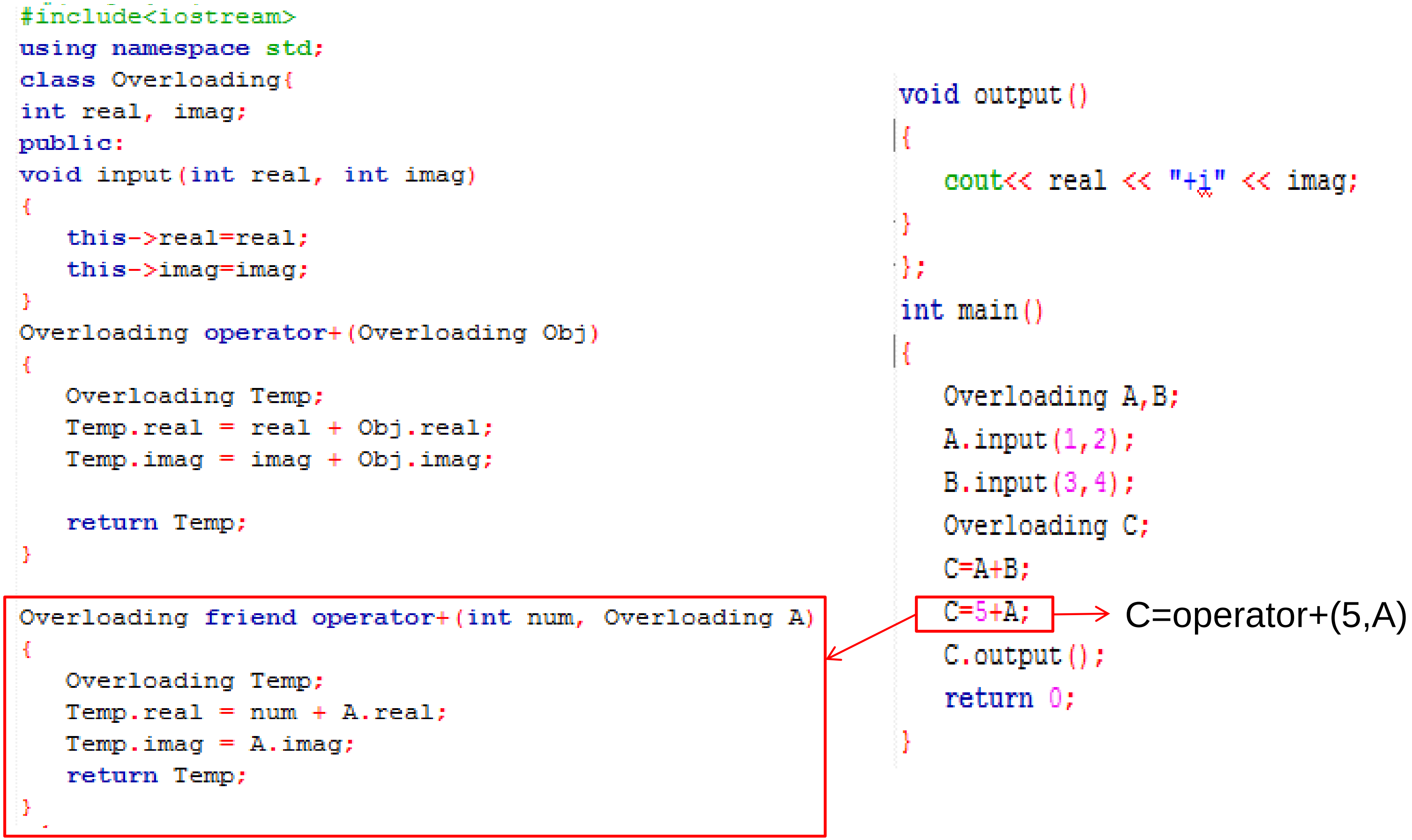
Overloading operators for operands of different types

```cpp
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}
Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

Overloading operator+(int num)
{
    Overloading Temp;
    Temp.real = real + num;
    Temp.imag = imag;
    return Temp;
}
```



```cpp
void output()
{
    cout<< real << "+i" << imag;
}
};
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A+B;
    C=5+A;
    C.output();
    return 0;
}
```

Overloading operators for operands of different types

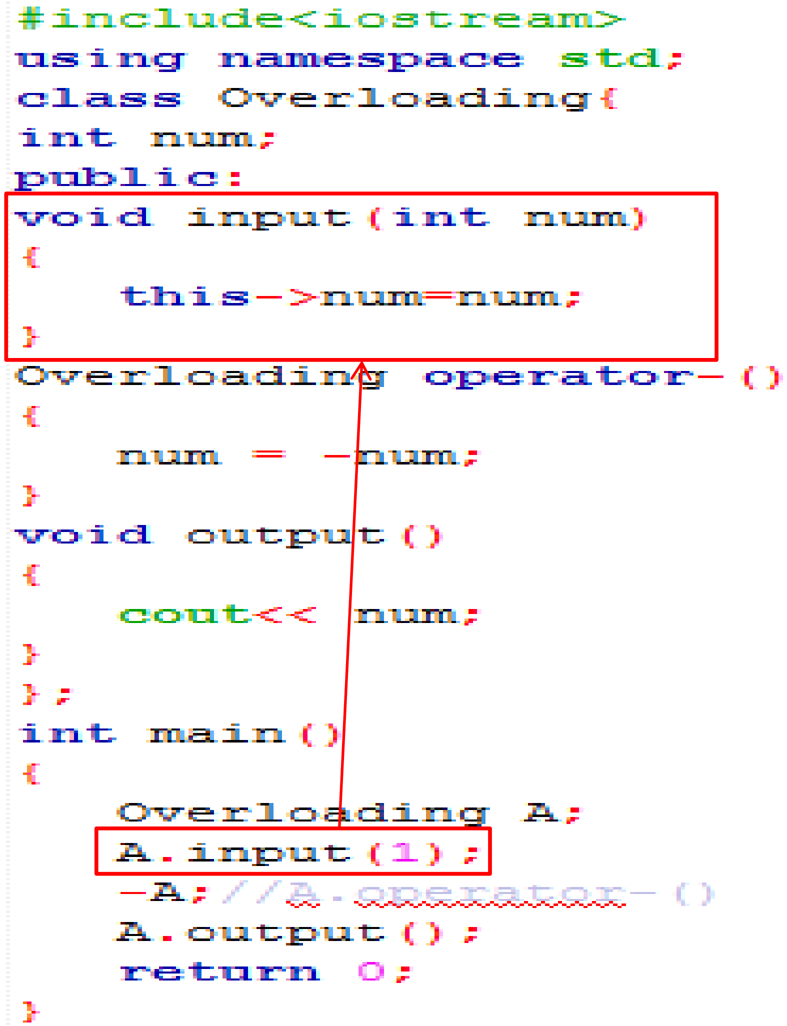


```
#include<iostream>
using namespace std;
class Overloading{
int real, imag;
public:
void input(int real, int imag)
{
    this->real=real;
    this->imag=imag;
}
Overloading operator+(Overloading Obj)
{
    Overloading Temp;
    Temp.real = real + Obj.real;
    Temp.imag = imag + Obj.imag;

    return Temp;
}

Overloading friend operator+(int num, Overloading A)
{
    Overloading Temp;
    Temp.real = num + A.real;
    Temp.imag = A.imag;
    return Temp;
}
```

```
void output()
{
    cout<< real << "+i" << imag;
}
};
int main()
{
    Overloading A,B;
    A.input(1,2);
    B.input(3,4);
    Overloading C;
    C=A+B;
    C=5+A;
    C.output();
    return 0;
}
```

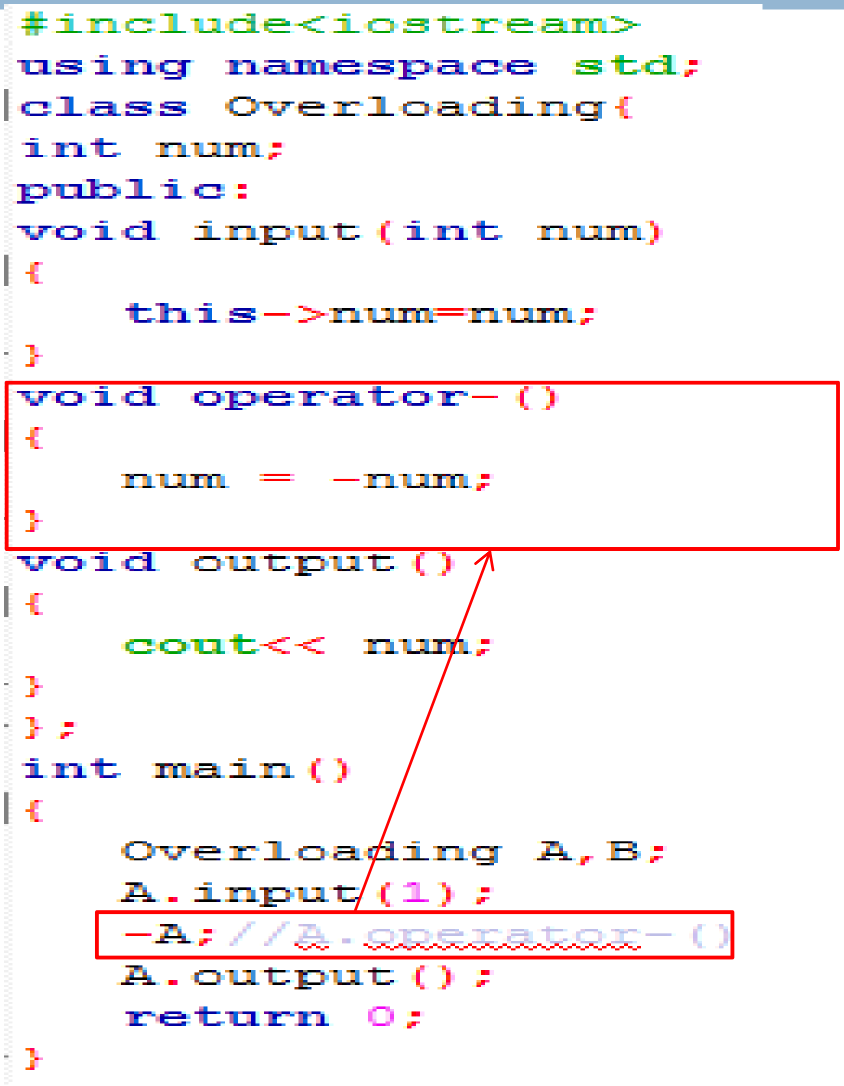
C=operator+(5,A)

Unary operator overloading



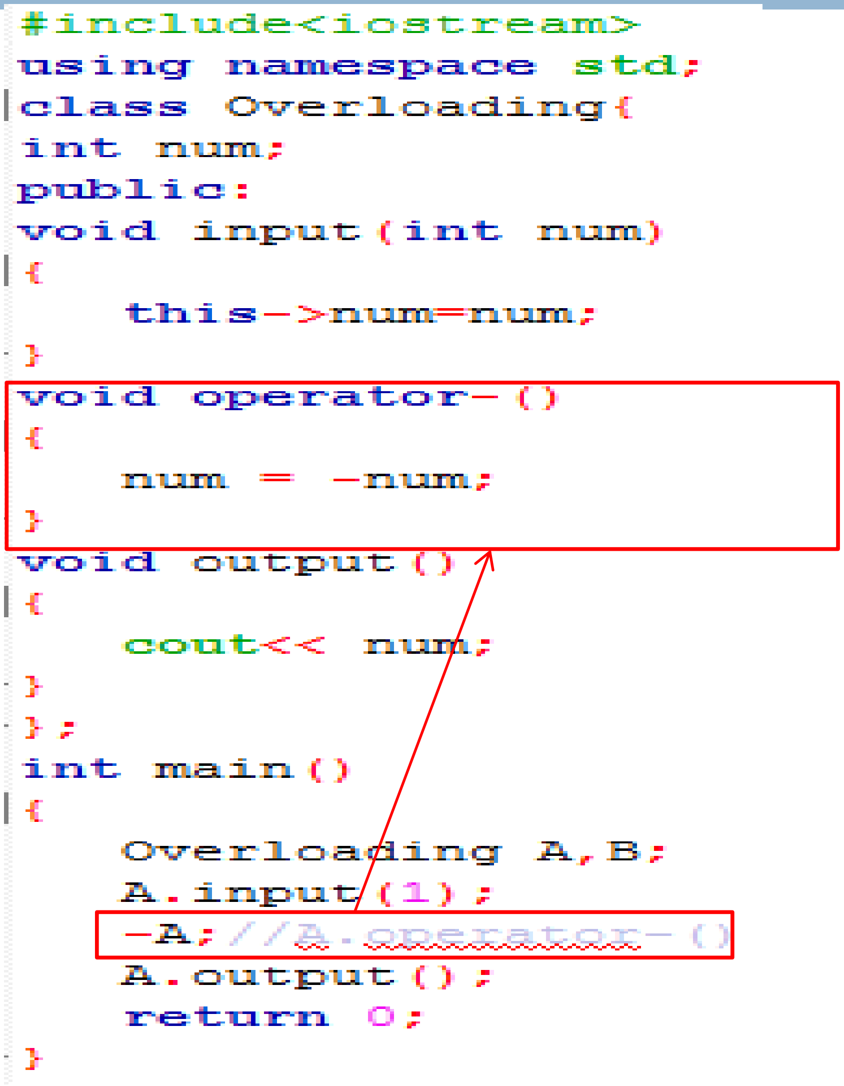
```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
Overloading operator-()
{
    num = -num;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A;
    A.input(1);
    -A; //A.operator-()
    A.output();
    return 0;
}
```

Unary operator overloading



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
     this->num=num;
}
void operator-()
{
     num = -num;
}
void output()
{
     cout<< num;
}
};
int main()
{
     Overloading A,B;
     A.input(1);
     -A;//A.operator-()
     A.output();
     return 0;
}
```

Unary operator overloading



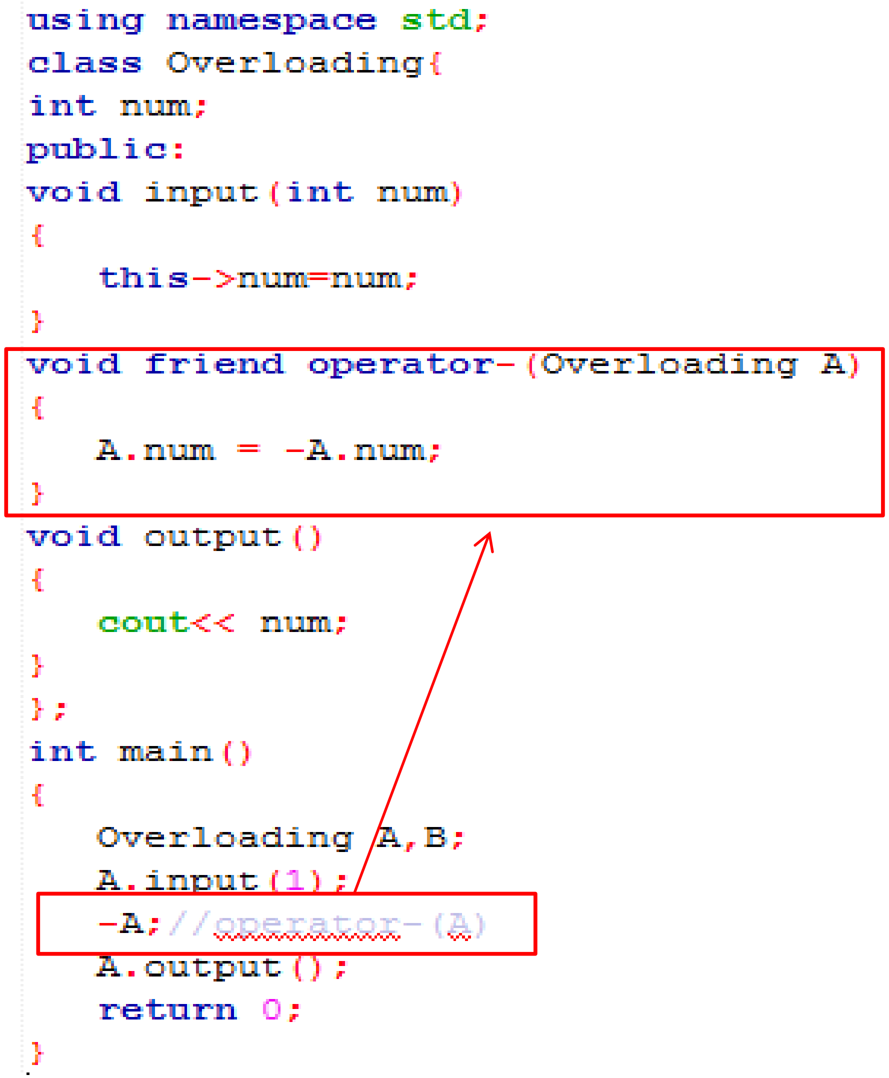
```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
void operator-()
{
    num = -num;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A,B;
    A.input(1);
    -A;//A.operator-()
    A.output();
    return 0;
}
```

Will B=-A work?

Yes!return *this;

B=-A;

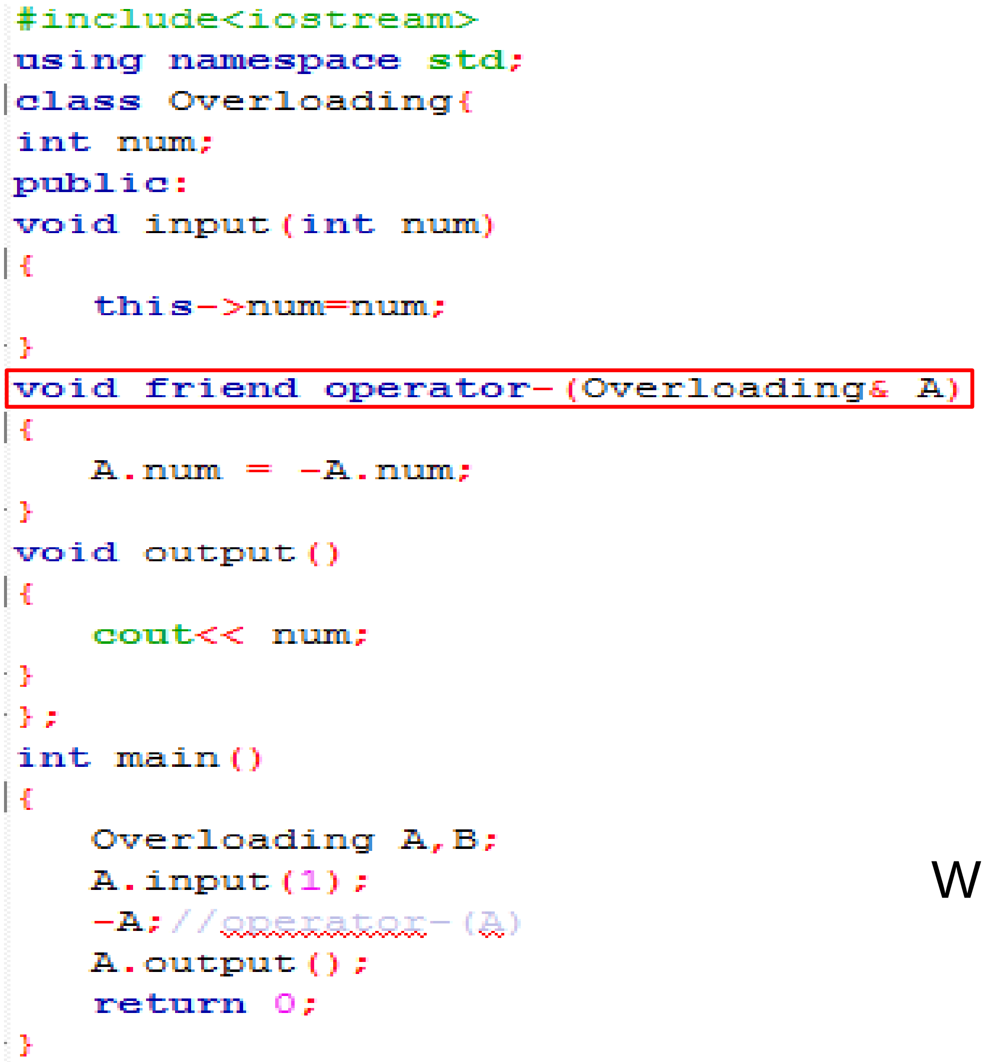
Unary operator overloading using friend function



```
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
void friend operator-(Overloading A)
{
    A.num = -A.num;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A,B;
    A.input(1);
    -A;//operator-(A)
    A.output();
    return 0;
}
```

What will be the output?

Unary operator overloading using friend function



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
void friend operator-(Overloading& A)
{
    A.num = -A.num;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A,B;
    A.input(1);
    -A;//operator-(A)
    A.output();
    return 0;
}
```

What about unary !(not) operator?

Overloading ++ and --

- Pre/post-incrementing/decrementing operators
  - Can be overloaded
  - How does the compiler distinguish between the two?
  - Prefix versions overloaded same as any other prefix unary operator would be. i.e. `d1.operator++();` for `++d1;`
- Postfix versions
  - When compiler sees postincrementing expression, such as

    ```
    d1++;
    ```

    Generates the member-function call

    ```
    d1.operator++( 0 );
    ```
  - Prototype:

    ```
    Date::operator++( int );
    ```

Overloading ++ and -- (Cont.)

- To distinguish prefix and postfix increment
  - Postfix increment has a dummy parameter
    - An `int` with value `0`
  - Prototype (member function)
    - `Date operator++( int );`
    - `d1++` becomes `d1.operator++( 0 )`
  - Prototype (global function)
    - `Date operator++( Date &, int );`
    - `d1++` becomes `operator++( d1, 0 )`

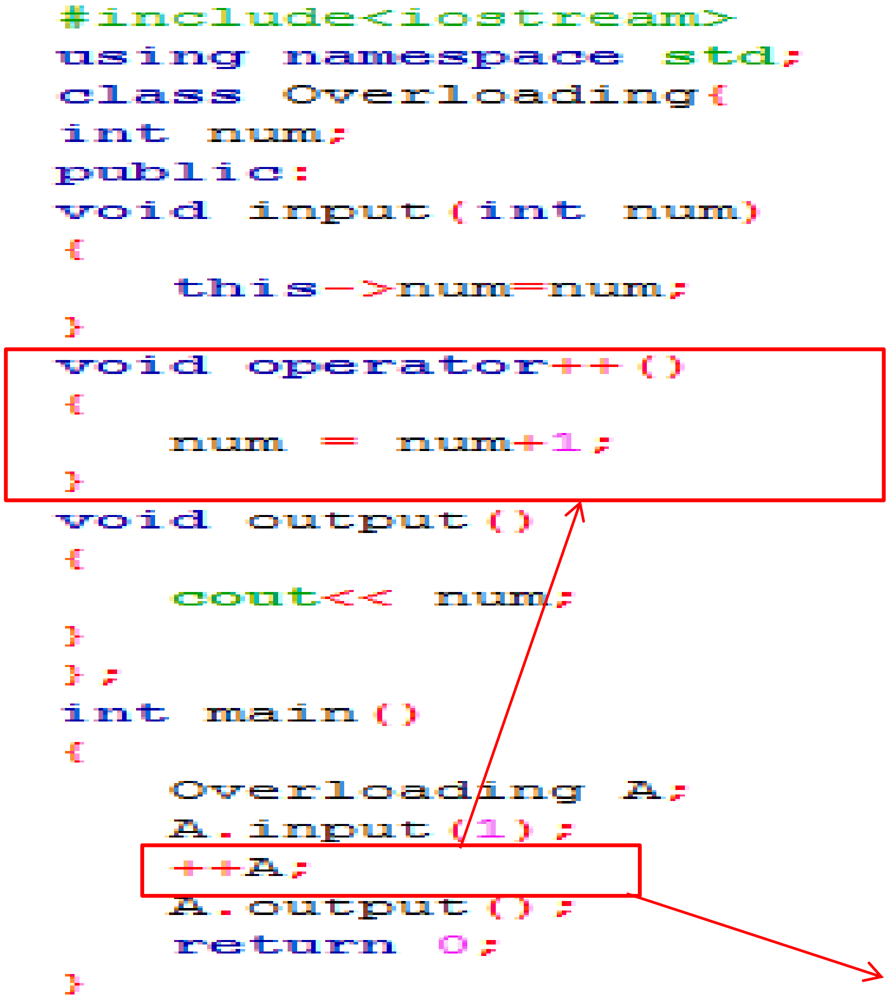
Overloading ++ and -- (Cont.)

- Return values
  - Prefix increment
    - Returns by reference (`Date &`)
    - *lvalue* (can be assigned)
  - Postfix increment
    - Returns by value
      - Returns temporary object with old value
    - *rvalue* (cannot be on left side of assignment)
- All this applies to decrement operators as well

Overloading ++ and -- (Cont.)

- The extra object that is created by the postfix increment (or decrement) operator can result in a significant performance problem—especially when the operator is used in a loop. For this reason, you should use the postfix increment (or decrement) operator only when the logic of the program requires postincrementing (or postdecrementing).

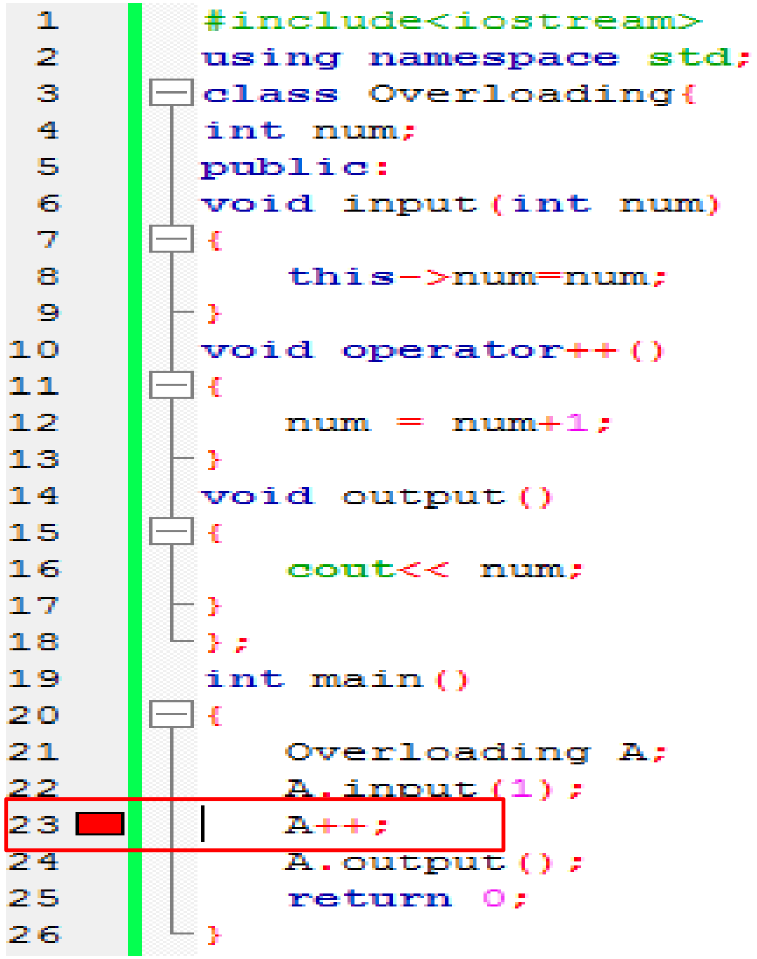
increment operator overloading



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
void operator++()
{
    num = num+1;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A;
    A.input(1);
    ++A;
    A.output();
    return 0;
}
```

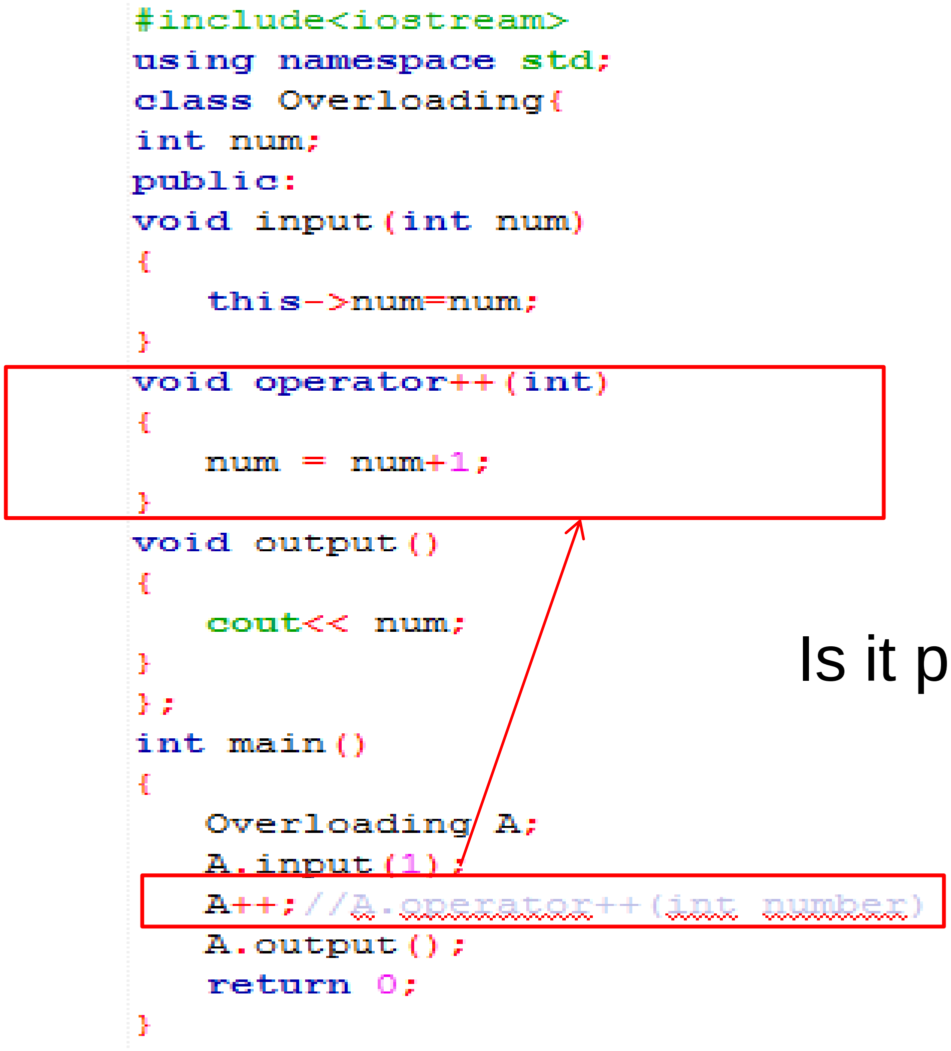
A.operator++()

increment operator overloading



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
void operator++()
{
    num = num+1;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A;
    A.input(1);
    A++;
    A.output();
    return 0;
}
```

increment operator overloading



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
void operator++(int)
{
    num = num+1;
}
void output()
{
    cout<< num;
}
};
int main()
{
    Overloading A;
    A.input(1);
    A++;//A.operator++(int number)
    A.output();
    return 0;
}
```

Is it possible to use friend function?

Case Study: A `Date` Class

- Example `Date` class
  - Overloaded increment operator
    - Change day, month and year
  - Overloaded += operator
  - Function to test for leap years
  - Function to determine if day is last of month

```cpp
// Fig. 11.12: Date.h
// Date class definition.
#ifndef DATE_H
#define DATE_H

#include <iostream>
using std::ostream;

class Date
{
   friend ostream &operator<<( ostream &, const Date & );
public:
   Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
   void setDate( int, int, int ); // set month, day, year
   Date &operator++(); // prefix increment operator
   Date operator++( int ); // postfix increment operator
   const Date &operator+=( int ); // add days, modify object
   bool leapYear( int ) const; // is date in a leap year?
   bool endOfMonth( int ) const; // is date at the end of month?
private:
   int month;
   int day;
   int year;

   static const int days[]; // array of days per month
   void helpIncrement(); // utility function for incrementing date
}; // end class Date

#endif
```

Date.h

(1 of 1)

Note the difference between prefix and postfix increment

```cpp
// Fig. 11.13: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h"

// initialize static member at file scope; one classwide copy
const int Date::days[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int m, int d, int y )
{
   setDate( m, d, y );
} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
} // end function setDate
```

Outline

Date.cpp

(1 of 4)

Date.cpp

(2 of 4)

```
// overloaded prefix increment operator
Date &Date::operator++()
{
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
} // end function operator++

// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
   Date temp = *this; // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
} // end function operator++

// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this; // enables cascading
} // end function operator+=
```

Postfix increment updates object and returns a copy of the original

Do not return a reference to `temp`, because it is a local variable that will be destroyed

```
// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear( int testYear ) const
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];
} // end function endOfMonth

```

Date.cpp

(3 of 4)

```cpp
// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      day++; // increment day
   else
      if ( month < 12 ) // day is end of month and month < 12
      {
         month++; // increment month
         day = 1; // first day of new month
      } // end if
      else // last day of year
      {
         year++; // increment year
         month = 1; // first month of new year
         day = 1; // first day of new month
      } // end else
} // end function helpIncrement

// overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static char *monthName[ 13 ] = { "", "January", "February",
      "March", "April", "May", "June", "July", "August",
      "September", "October", "November", "December" };
   output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
   return output; // enables cascading
} // end function operator<<
```

Outline

Date.cpp

(4 of 4)

```
// Fig. 11.14: fig11_14.cpp
// Date class test program.
#include <iostream>
using std::cout;
using std::endl;

#include "Date.h" // Date class definition

int main()
{
   Date d1; // defaults to January 1, 1900
   Date d2( 12, 27, 1992 ); // December 27, 1992
   Date d3( 0, 99, 8045 ); // invalid date

   cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
   cout << "\n\nd2 += 7 is " << ( d2 += 7 );

   d3.setDate( 2, 28, 1992 );
   cout << "\n\n  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";

   Date d4( 7, 13, 2002 );

   cout << "\n\nTesting the prefix increment operator:\n"
      << "  d4 is " << d4 << endl;
   cout << "++d4 is " << ++d4 << endl;
   cout << "  d4 is " << d4;

```

fig11_14.cpp

(1 of 2)

Demonstrate prefix increment

```
   cout << "\n\nTesting the postfix increment operator:\n"
      << "  d4 is " << d4 << endl;
   cout << "d4++ is " << d4++ << endl;
   cout << "  d4 is " << d4 << endl;
   return 0;
} // end main
```

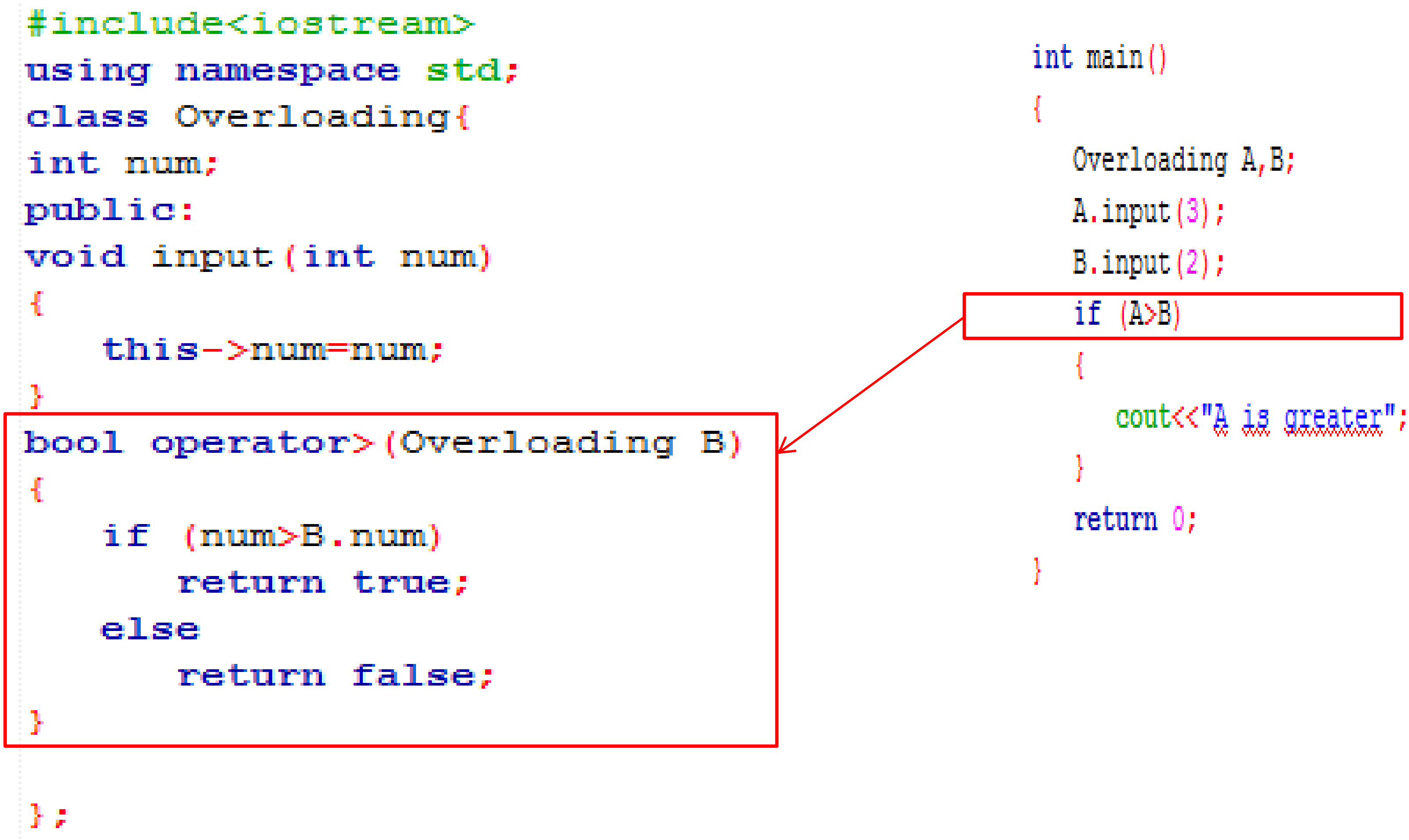
Demonstrate postfix increment

fig11_14.cpp

(2 of 2)

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

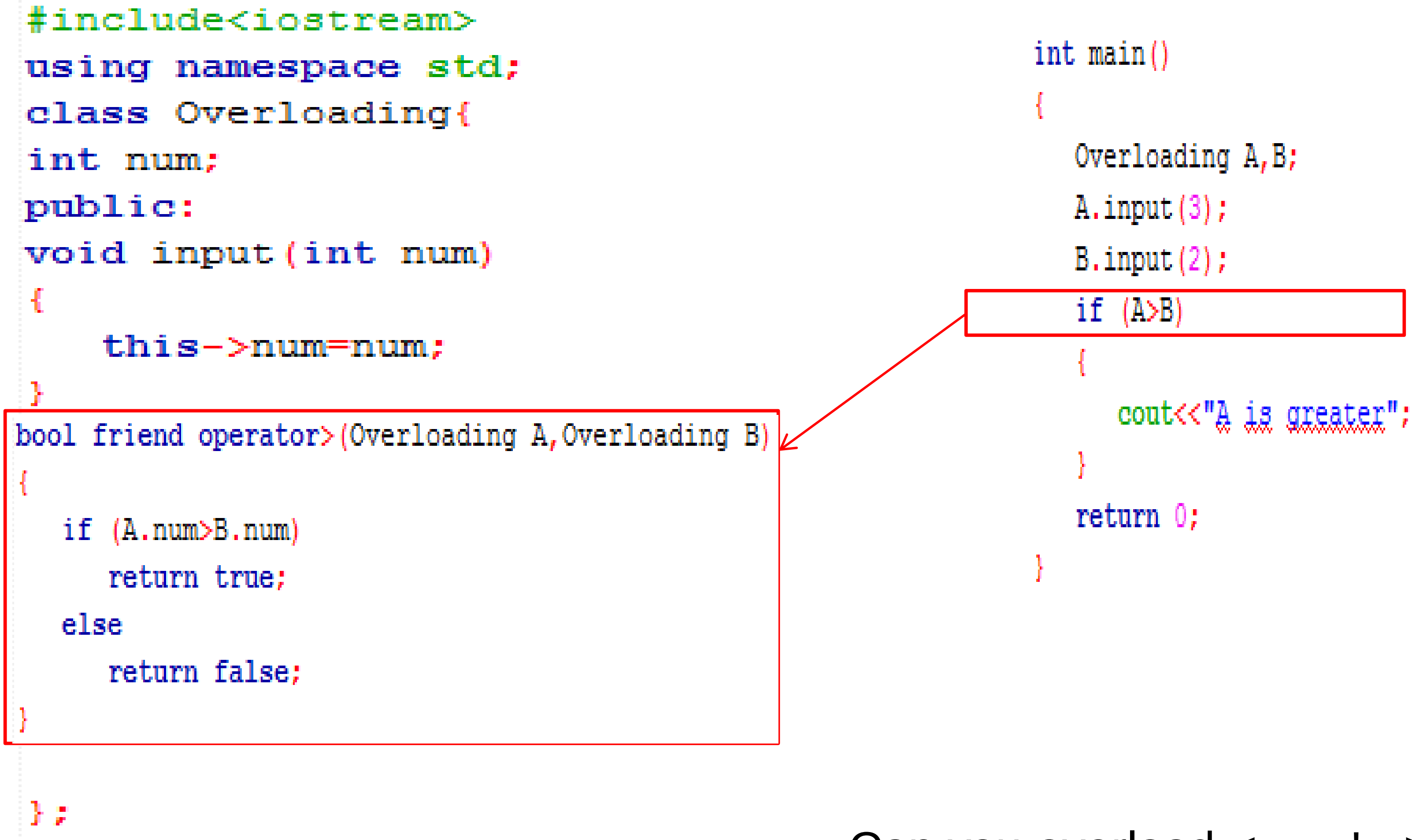
Overloading the comparison operators



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
bool operator>(Overloading B)
{
    if (num>B.num)
        return true;
    else
        return false;
}

};
```

```
int main()
{
    Overloading A,B;
    A.input(3);
    B.input(2);
    if (A>B)
    {
        cout<<"A is greater";
    }
    return 0;
}
```

Overloading the comparison operators using friend function



```
#include<iostream>
using namespace std;
class Overloading{
int num;
public:
void input(int num)
{
    this->num=num;
}
bool friend operator>(Overloading A,Overloading B)
{
   if (A.num>B.num)
       return true;
   else
       return false;
}

};
```

```
int main()
{
    Overloading A,B;
    A.input(3);
    B.input(2);
    if (A>B)
    {
        cout<<"A is greater";
    }
    return 0;
}
```

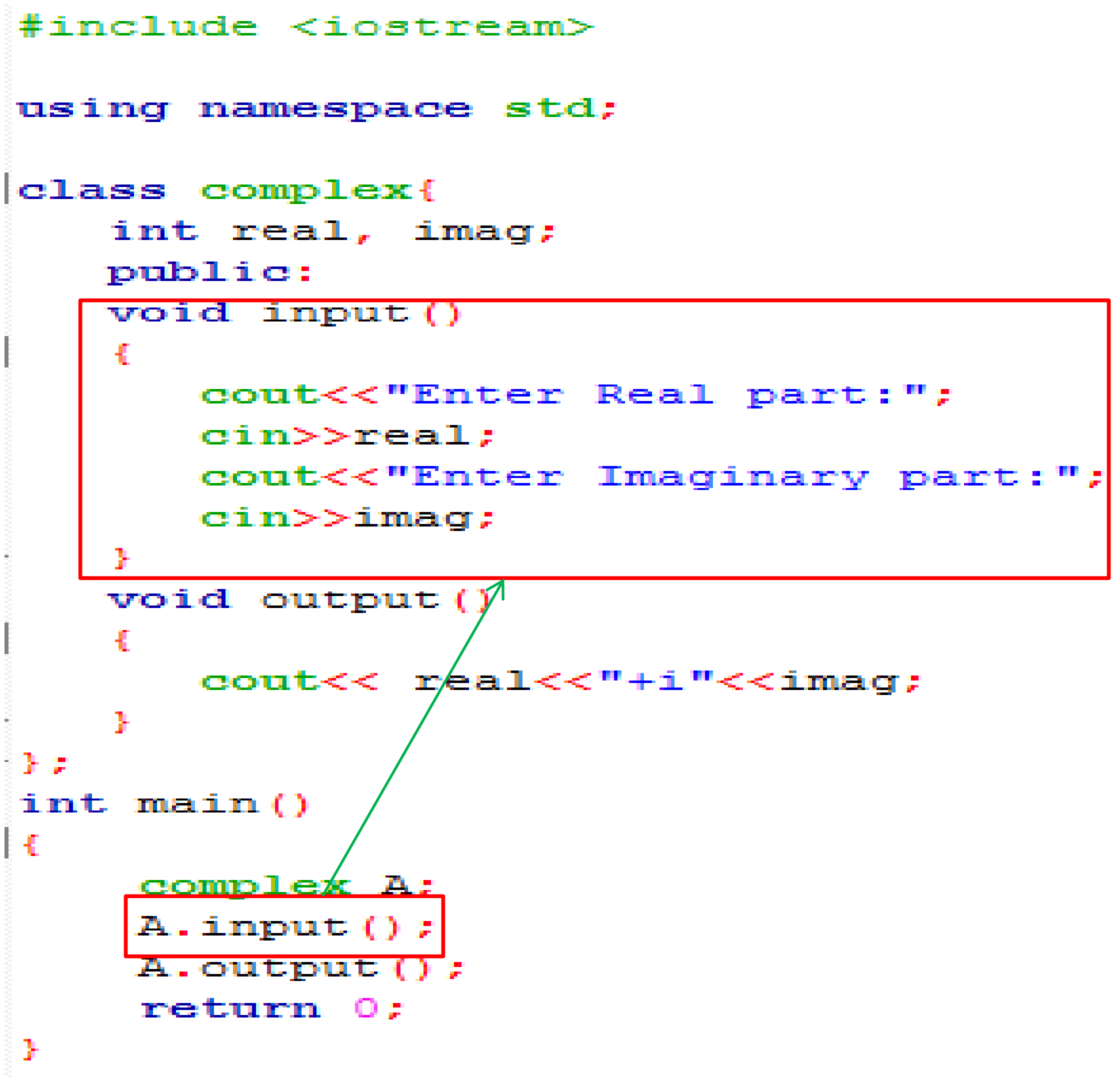
Can you overload <, ==,!=,>=,<= ?

References

- http://www.learncpp.com
- Object oriented programming with c++ -- E Balagurusamy

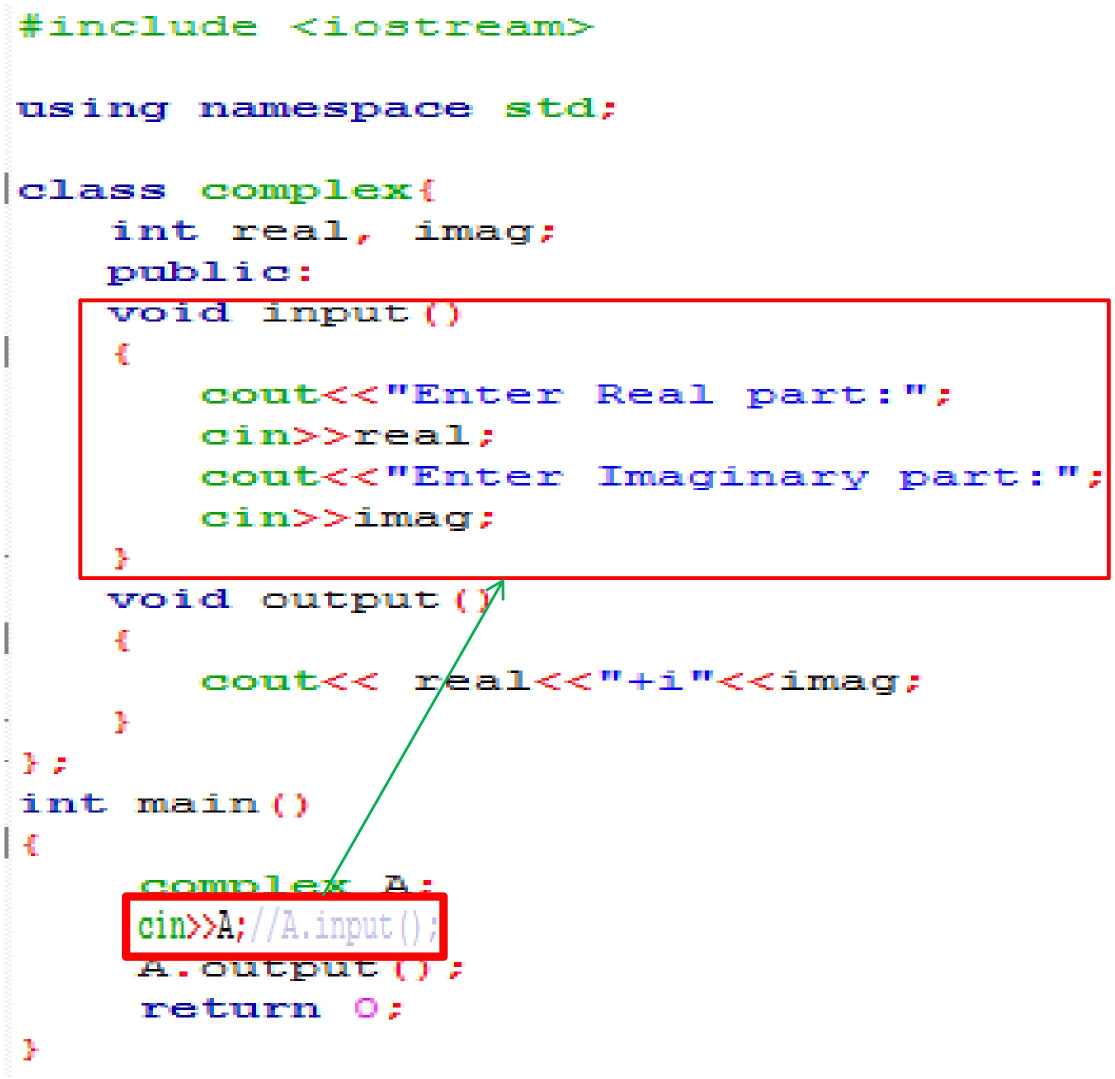
OPERATOR OVERLOADING

Overloading IO operator



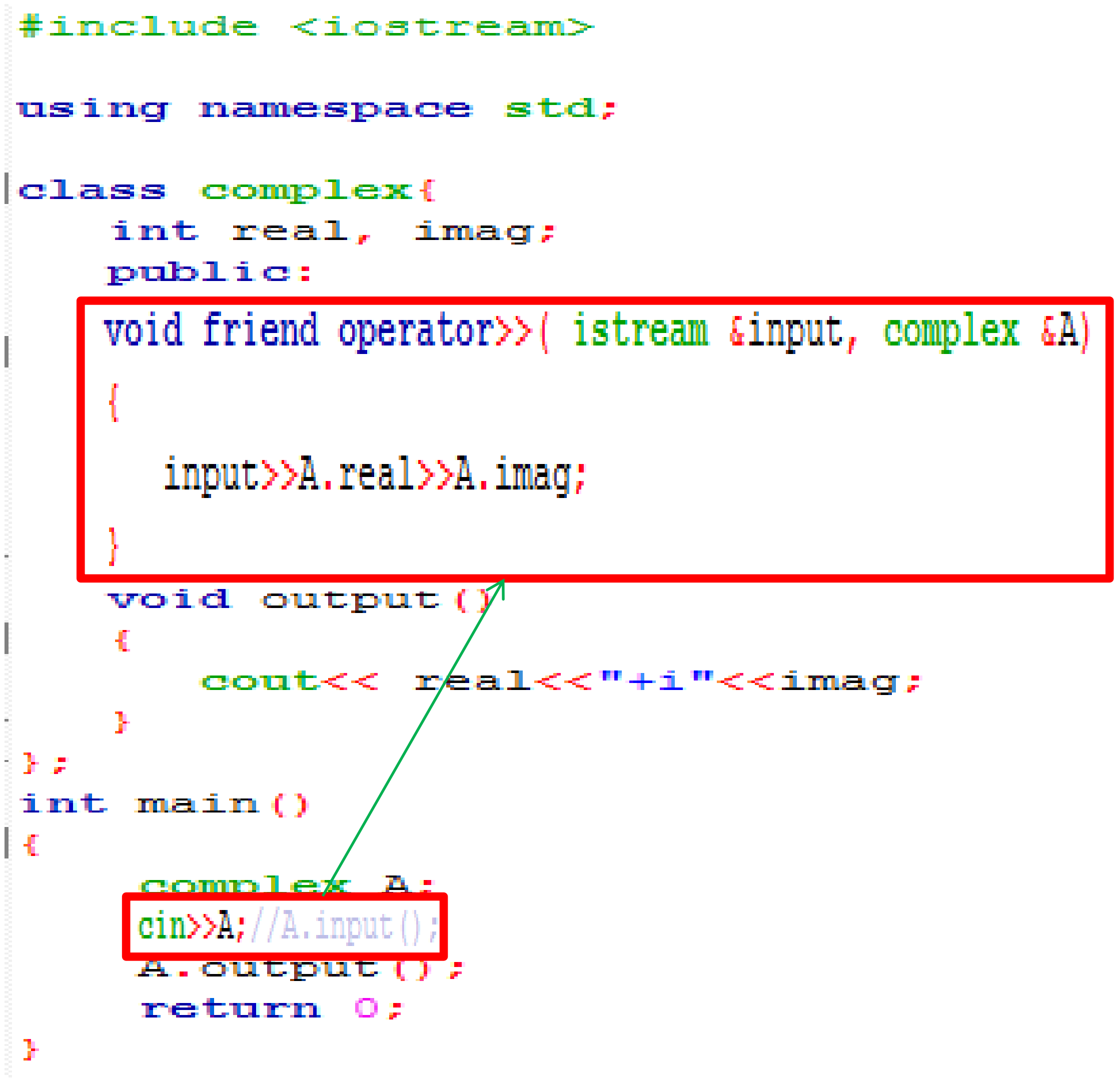
```
#include <iostream>

using namespace std;

class complex{
    int real, imag;
    public:
    void input()
    {
        cout<<"Enter Real part:";
        cin>>real;
        cout<<"Enter Imaginary part:";
        cin>>imag;
    }
    void output()
    {
        cout<< real<<"+i"<<imag;
    }
};
int main()
{
    complex A;
    A.input();
    A.output();
    return 0;
}
```

Overloading IO operator



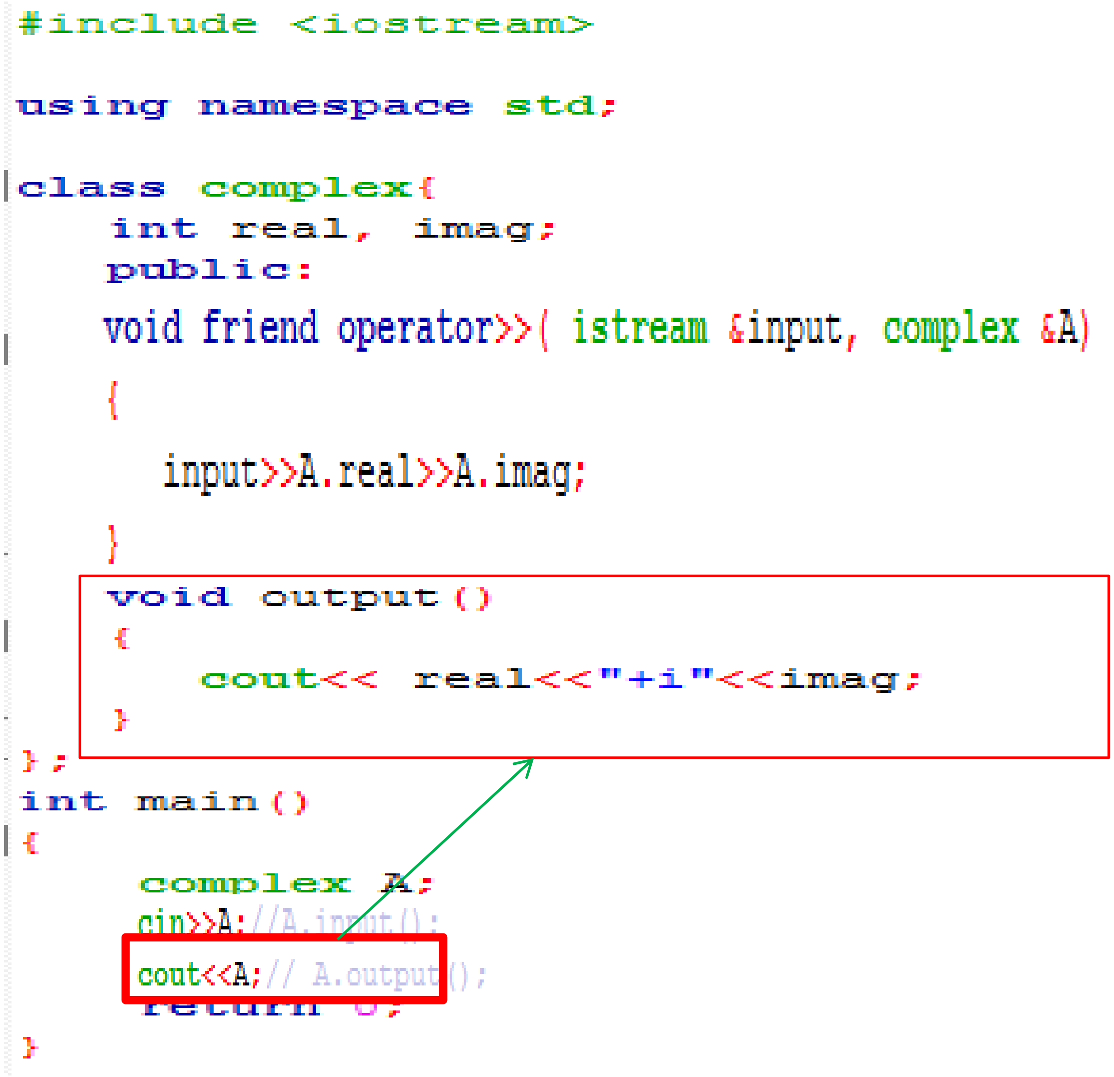
```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
     public:
     void input()
     {
          cout<<"Enter Real part:";
          cin>>real;
          cout<<"Enter Imaginary part:";
          cin>>imag;
     }
     void output()
     {
          cout<< real<<"+i"<<imag;
     }
};
int main()
{
      complex A;
      cin>>A;//A.input();
      A.output();
      return 0;
}
```

Overloading IO operator



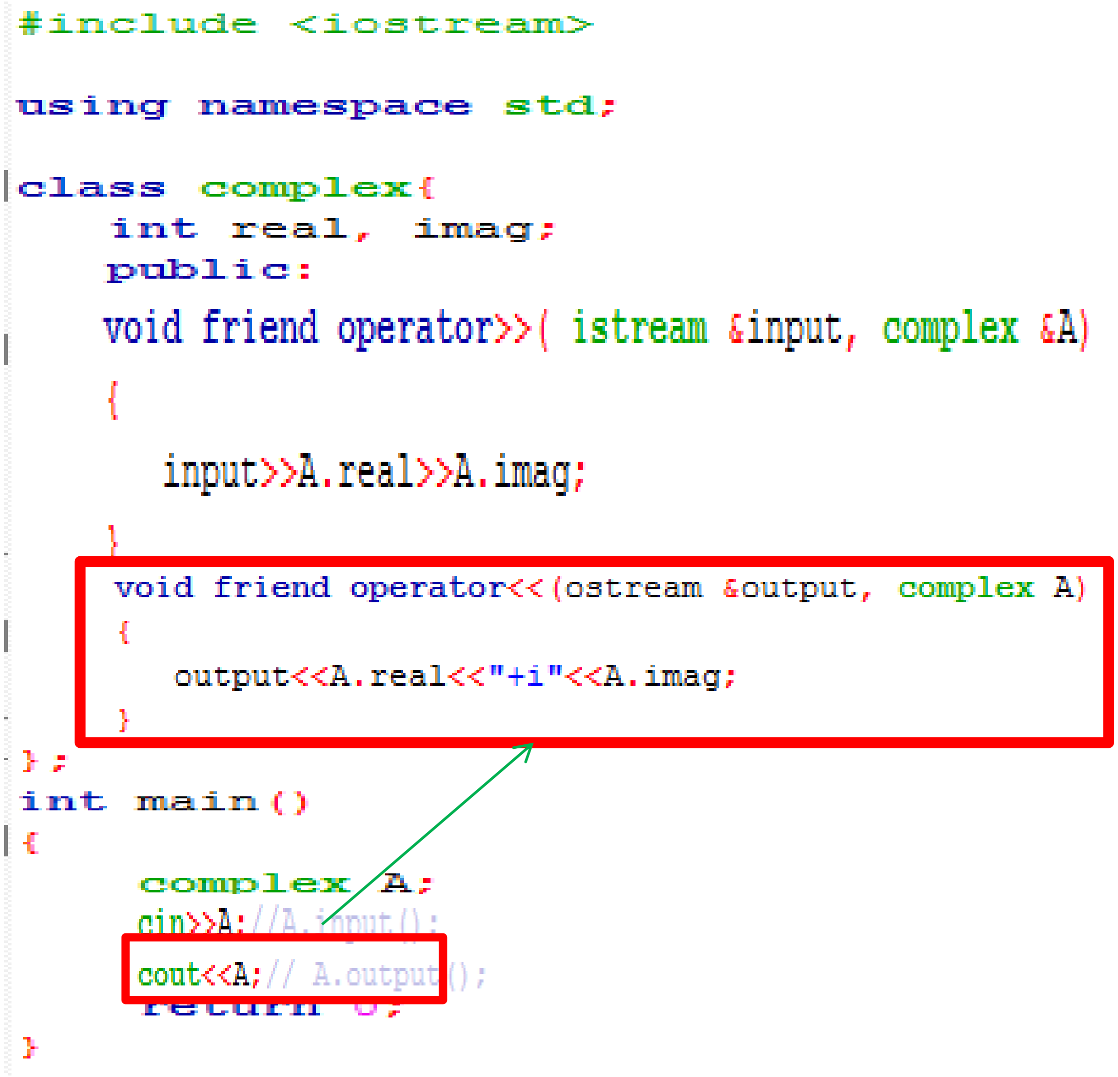
```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
     public:
     void friend operator>>( istream &input, complex &A)
     {
        input>>A.real>>A.imag;
     }
     void output()
     {
          cout<< real<<"+i"<<imag;
     }
};
int main()
{
      complex A;
      cin>>A;//A.input();
      A.output();
      return 0;
}
```

Overloading IO operator



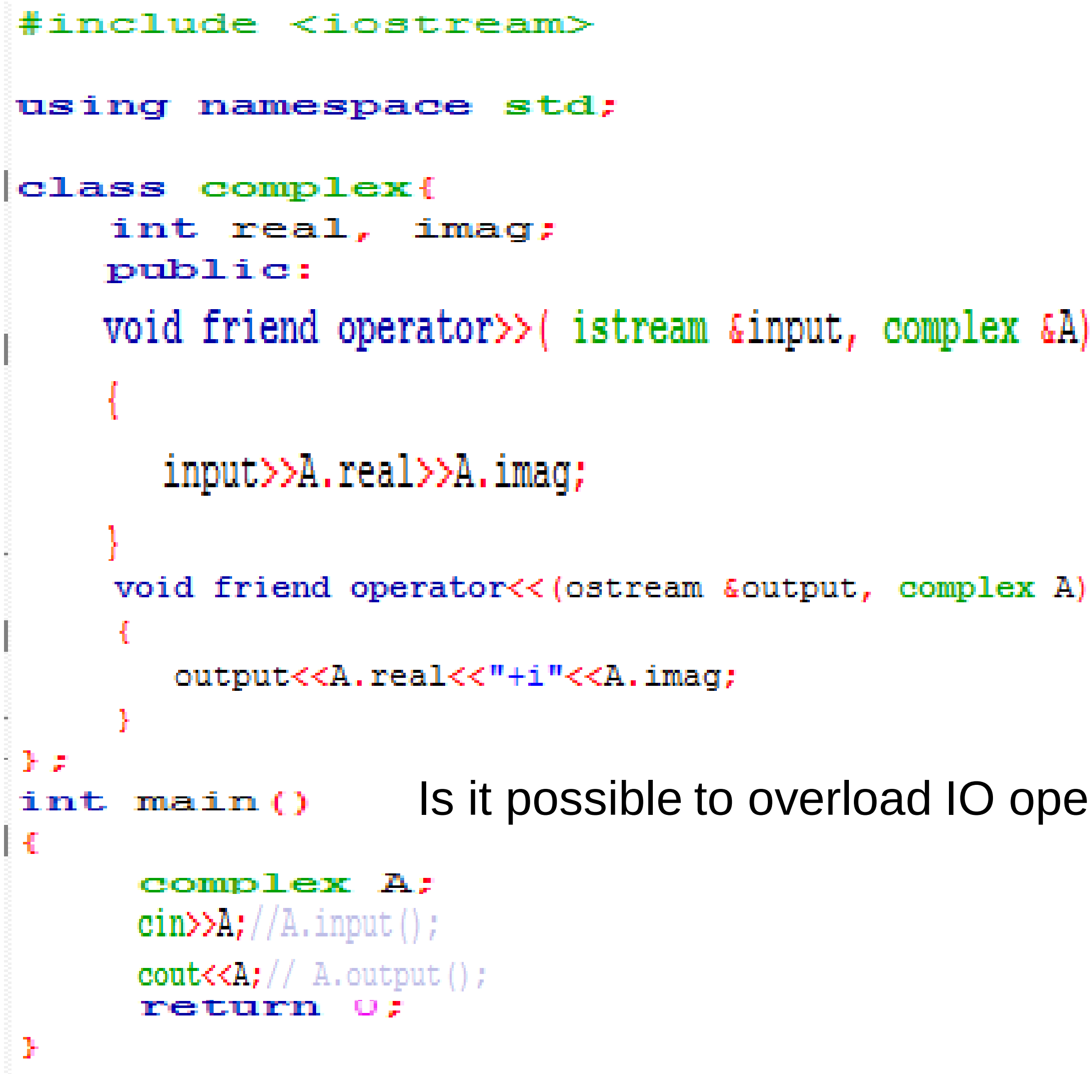
```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
     public:
     void friend operator>>( istream &input, complex &A)
     {
        input>>A.real>>A.imag;
     }
     void output()
     {
          cout<< real<<"+i"<<imag;
     }
};
int main()
{
       complex A;
       cin>>A;//A.input();
       cout<<A;// A.output();
       return 0;
}
```

Overloading IO operator



```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
    public:
    void friend operator>>( istream &input, complex &A)
    {
       input>>A.real>>A.imag;
    }
    void friend operator<<(ostream &output, complex A)
    {
       output<<A.real<<"+i"<<A.imag;
    }
};
int main()
{
      complex A;
      cin>>A;//A.input();
      cout<<A;// A.output();
      return 0;
}
```

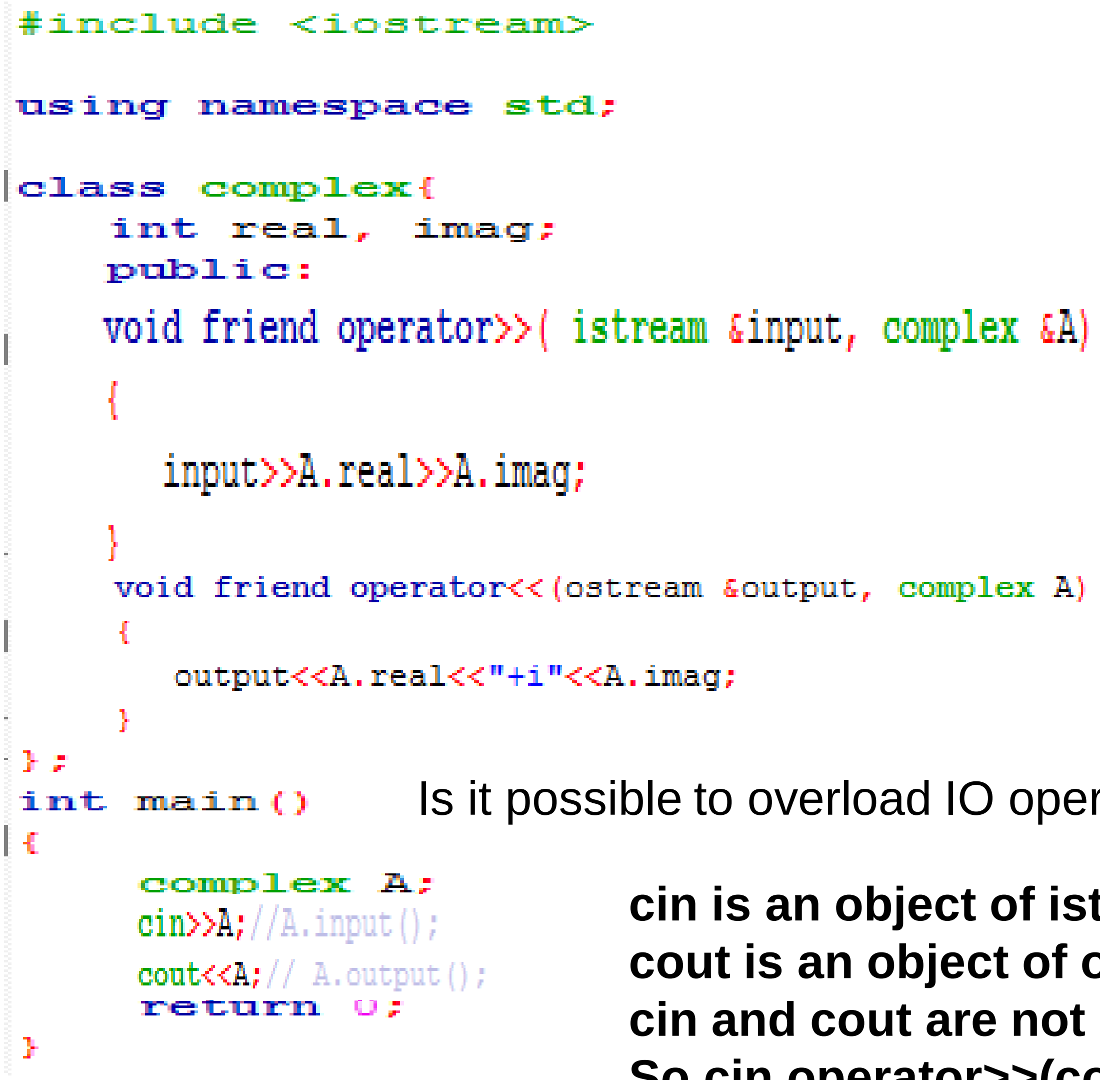
Overloading IO operator



```
#include <iostream>

using namespace std;

class complex{
      int real, imag;
     public:
     void friend operator>>( istream &input, complex &A)
     {
        input>>A.real>>A.imag;
     }
      void friend operator<<(ostream &output, complex A)
      {
         output<<A.real<<"+i"<<A.imag;
      }
};
int main()
{
      complex A;
      cin>>A;//A.input();
      cout<<A;// A.output();
      return 0;
}
```

Is it possible to overload IO operator using member function?

Overloading IO operator

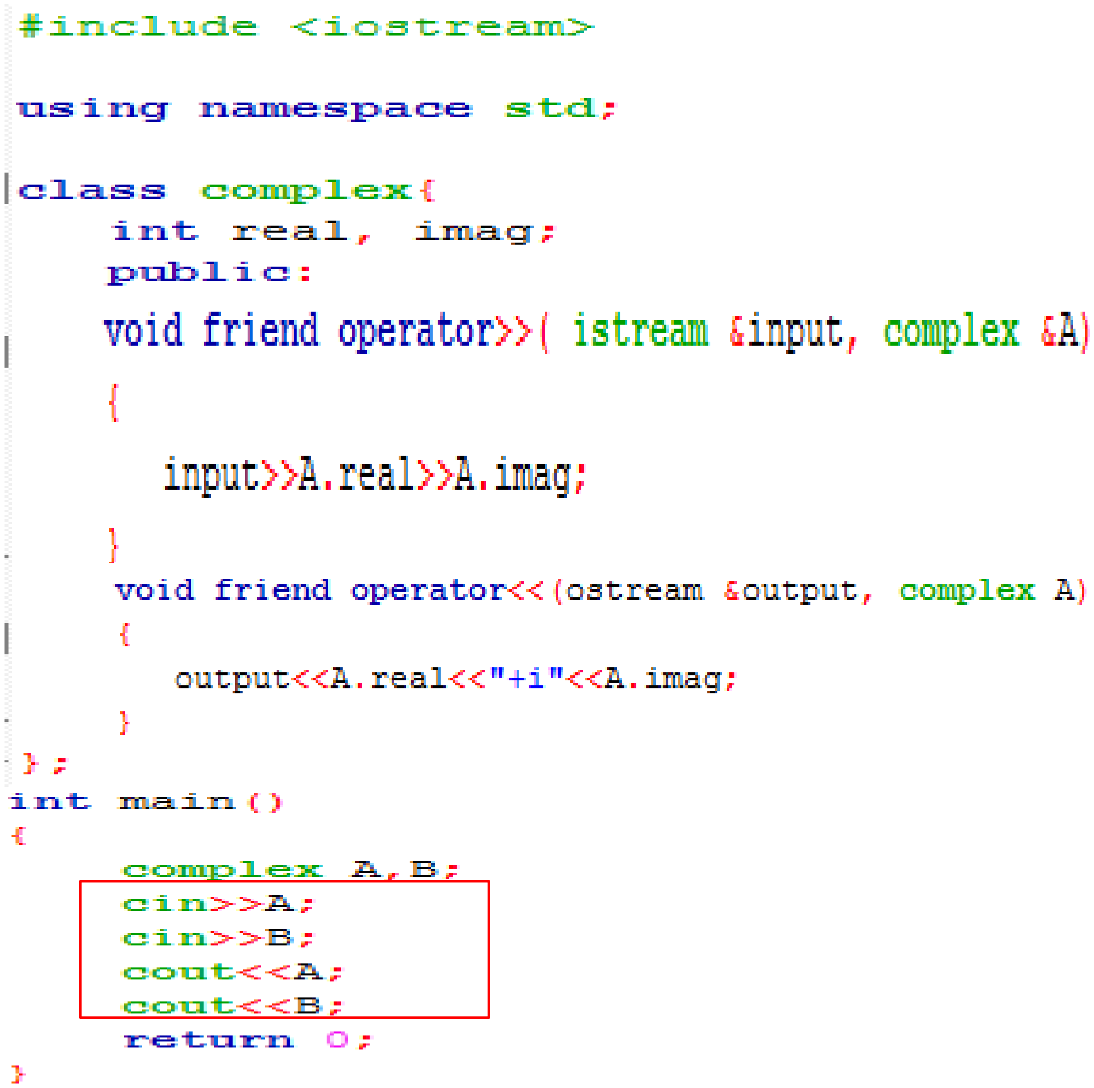


```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
    public:
    void friend operator>>( istream &input, complex &A)
    {
       input>>A.real>>A.imag;
    }
     void friend operator<<(ostream &output, complex A)
     {
        output<<A.real<<"+i"<<A.imag;
     }
};
int main()
{
      complex A;
      cin>>A;//A.input();
      cout<<A;// A.output();
      return 0;
}
```

Is it possible to overload IO operator using member function?

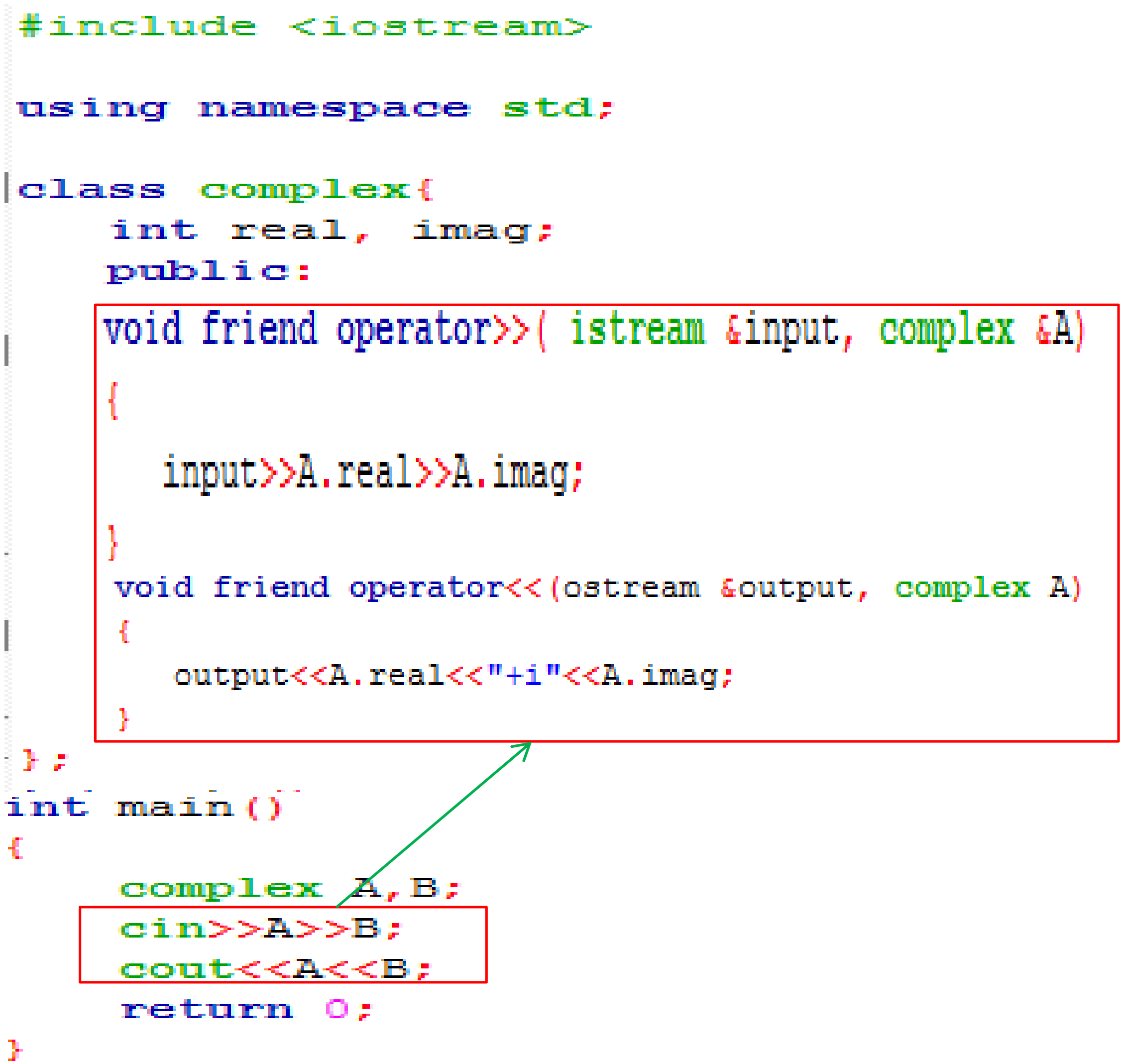
cin is an object of istream class.
cout is an object of ostream class.
cin and cout are not objects of complex class.
So cin.operator>>(complex A) is not possible

Overloading IO operator



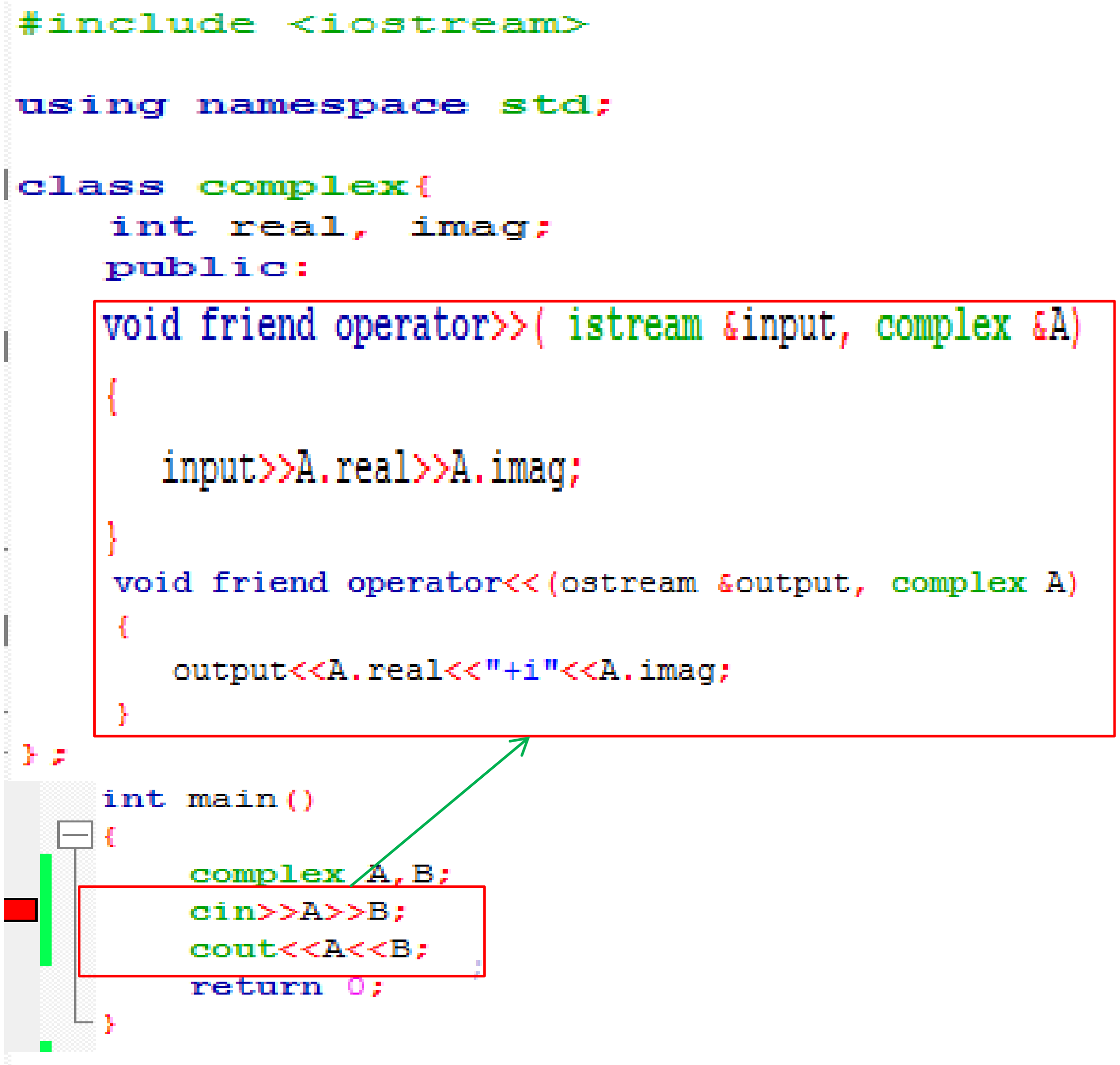
```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
    public:
    void friend operator>>( istream &input, complex &A)
    {
       input>>A.real>>A.imag;
    }
     void friend operator<<(ostream &output, complex A)
     {
        output<<A.real<<"+i"<<A.imag;
     }
};
int main()
{
      complex A,B;
      cin>>A;
      cin>>B;
      cout<<A;
      cout<<B;
      return 0;
}
```

Overloading IO operator



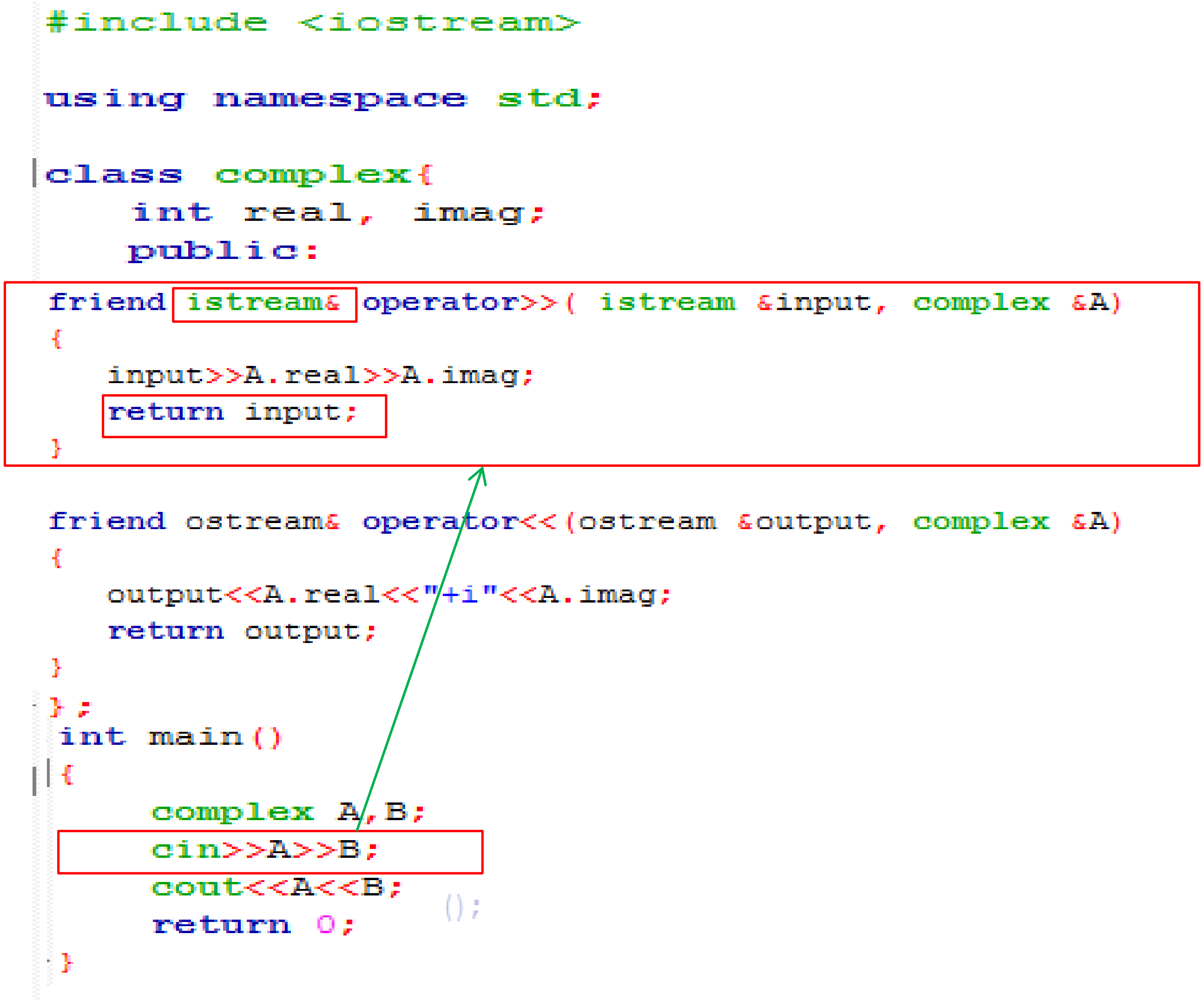
```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
     public:
     void friend operator>>( istream &input, complex &A)
     {
        input>>A.real>>A.imag;
     }
     void friend operator<<(ostream &output, complex A)
     {
        output<<A.real<<"+i"<<A.imag;
     }
};
int main()
{
      complex A,B;
      cin>>A>>B;
      cout<<A<<B;
      return 0;
}
```

Overloading IO operator



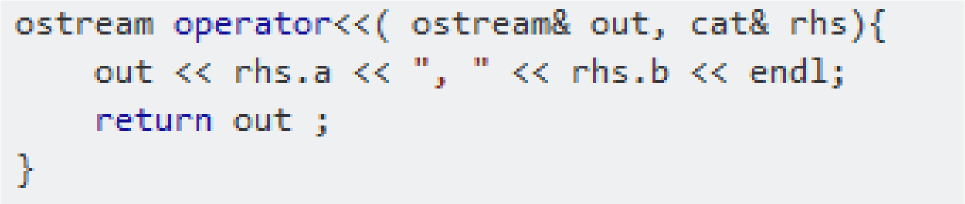
```
#include <iostream>

using namespace std;

class complex{
      int real, imag;
      public:
      void friend operator>>( istream &input, complex &A)
      {
         input>>A.real>>A.imag;
      }
      void friend operator<<(ostream &output, complex A)
      {
         output<<A.real<<"+i"<<A.imag;
      }
};
int main()
{
        complex A,B;
        cin>>A>>B;
        cout<<A<<B;
        return 0;
}
```

Overloading IO operator

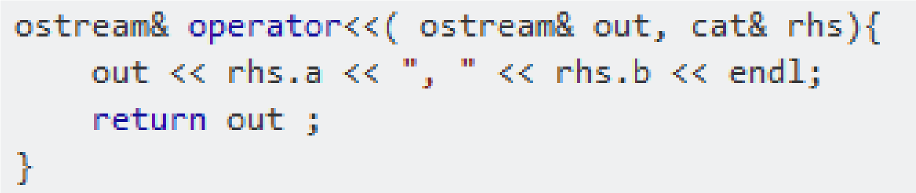


```
#include <iostream>

using namespace std;

class complex{
     int real, imag;
     public:
friend istream& operator>>( istream &input, complex &A)
{
    input>>A.real>>A.imag;
    return input;
}

friend ostream& operator<<(ostream &output, complex &A)
{
    output<<A.real<<"+i"<<A.imag;
    return output;
}
};
 int main()
 {
       complex A,B;
       cin>>A>>B;
       cout<<A<<B;
       return 0;
 }
```

why overloading of operator<< must return by reference?



```
ostream operator<<( ostream& out, cat& rhs){
    out << rhs.a << ", " << rhs.b << endl;
    return out ;
}
```



```
ostream& operator<<( ostream& out, cat& rhs){
    out << rhs.a << ", " << rhs.b << endl;
    return out ;
}
```

- ❑ In the first example, you return a copy of the stream object which is not allowed because copy-constructor (and copy-assignment as well) of the all stream classes in C++ is disabled by having them made private.
- ❑ Since you cannot make a copy of an stream object, you're required to return it by reference, which you're doing in the second example which is why it is working fine.
- ❑ You may choose to return nothing at all (i.e you can make the return type void), but if you do so, then you would not be able to chain as stream << a << b.
- ❑ You've to write them separately as stream <<a and then stream << b.

If you want to know why copying of stream objects is disabled

- Copying of ANY stream in C++ is disabled by having made the copy constructor private.
- Any means ANY, whether it is stringstream, istream, ostream,iostream or whatever.
- Copying of stream is disabled because it doesn't make sense. Its very very very important to understand what stream means, to actually understand why copying stream does not make sense. stream is not a container that you can make copy of. It doesn't contain data.

If you want to know why copying of stream objects is disabled

- If a list/vector/map or any container is a bucket, then stream is a hose through which data *flows*. Think of stream as some pipe *through* which you get data; a pipe - at one side is the source (sender), on the other side is the sink (receiver). That is called unidirectional stream. There're also bidirectional streams through which data *flows* in both direction. So what does it make sense making a copy of such a thing? It doesn't contain any data at all. It is *through* which you get data.

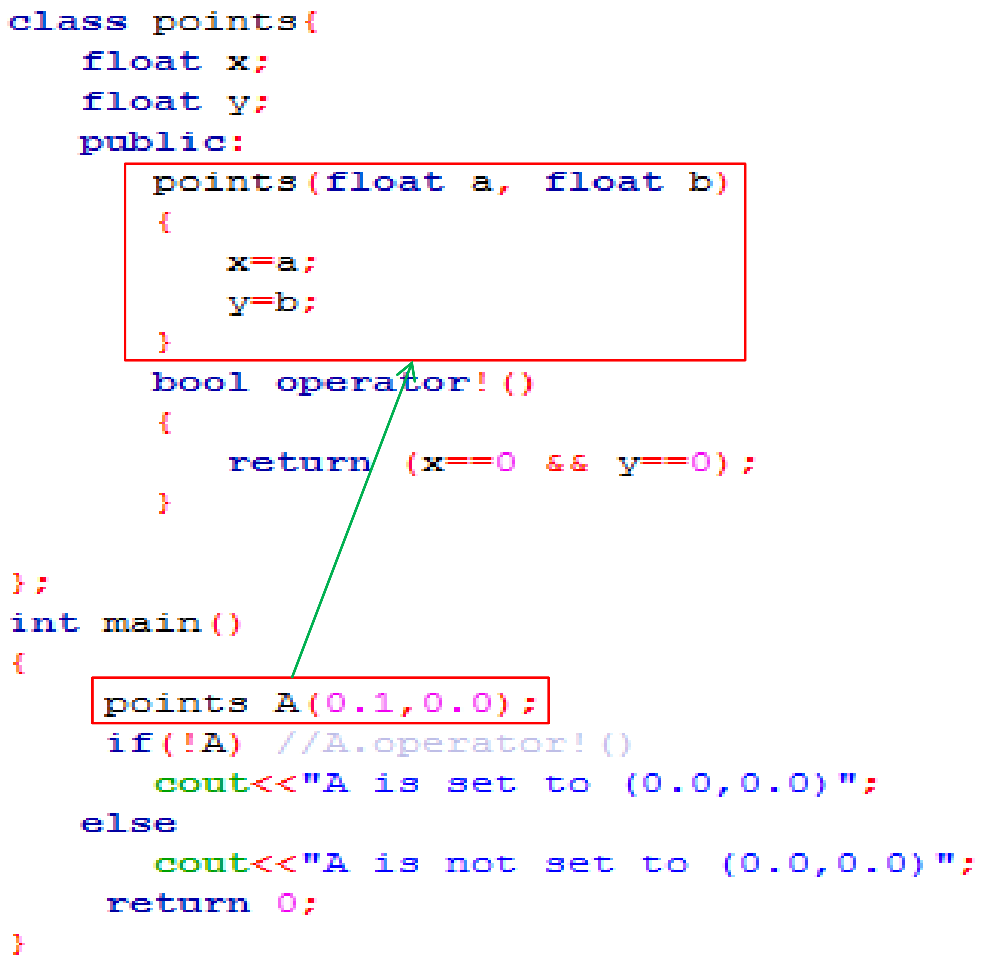
If you want to know why copying of stream objects is disabled

- Now suppose for a while if making a copy of stream is allowed, and you created a copy of std::cin which is in fact input stream. Say the copied object is copy_cin.
- Now ask yourself : does it make sense to read data from copy_cin stream when the very same data has already been read from std::cin.
- No, it doesn't make sense, because the user entered the data only once, the keyboard (or the input device) generated the electric signals only once and they flowed through all other hardwares and low-level APIs only once.
- How can your program read it twice or more?

If you want to know why copying of stream objects is disabled

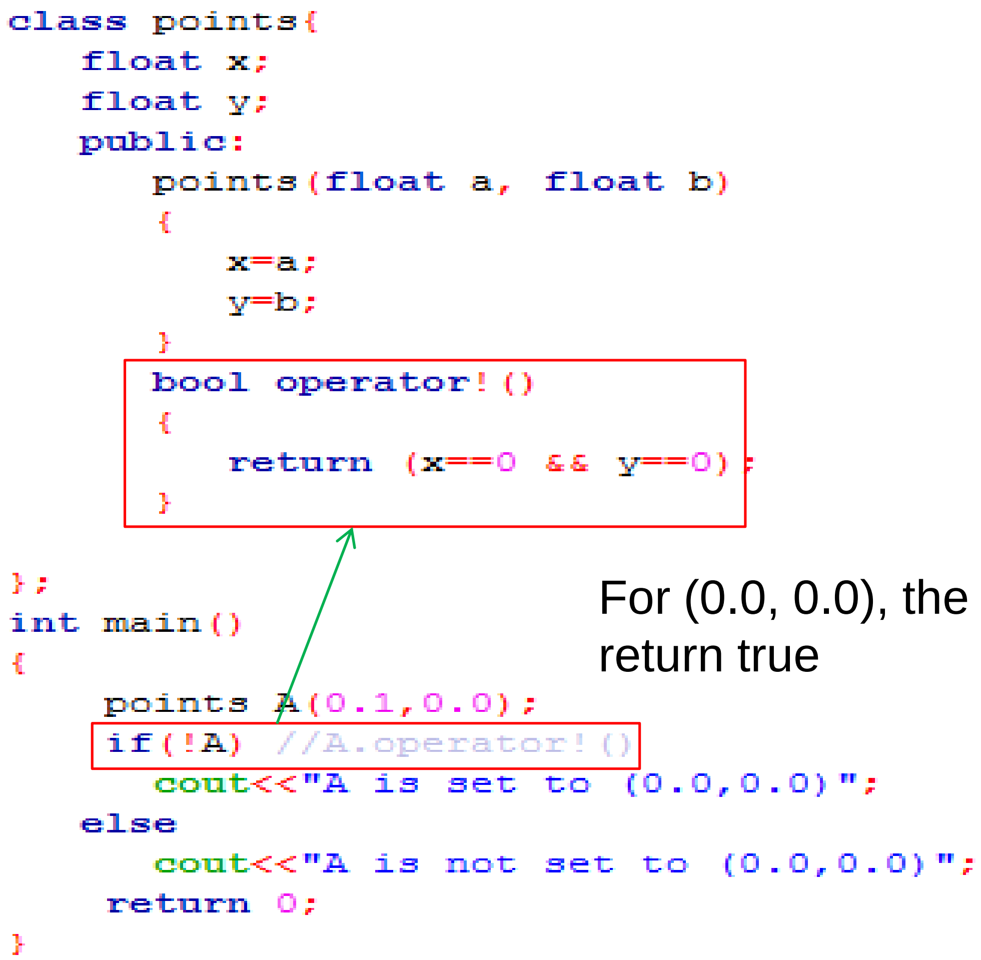
- Hence, creating copy is not allowed, but creating reference is allowed:
- std::istream copy_cin = std::cin; //error
- std::istream & ref_cin = std::cin; //ok
- Also note that you can create another instance of stream and can make it use the same underlying buffer which the old stream is currently using

! Operator overloading



```
class points{
   float x;
   float y;
   public:
      points(float a,  float b)
      {
         x=a;
         y=b;
      }
      bool operator!()
      {
         return (x==0 && y==0);
      }

};
int main()
{
    points A(0.1,0.0);
    if(!A) //A.operator!()
      cout<<"A is set to (0.0,0.0)";
   else
      cout<<"A is not set to (0.0,0.0)";
    return 0;
}
```

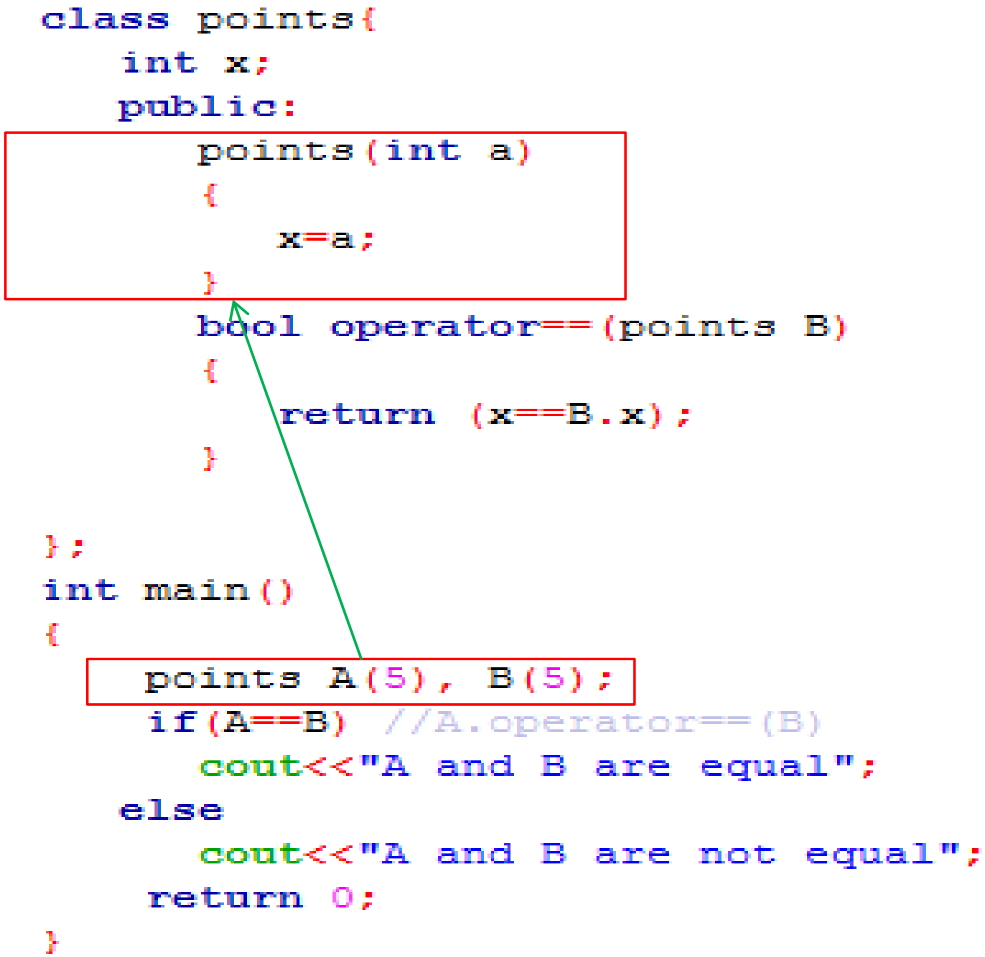
! Operator overloading



```
class points{
   float x;
   float y;
   public:
      points(float a, float b)
      {
         x=a;
         y=b;
      }
      bool operator!()
      {
         return (x==0 && y==0);
      }

};
int main()
{
    points A(0.1,0.0);
    if(!A) //A.operator!()
      cout<<"A is set to (0.0,0.0)";
   else
      cout<<"A is not set to (0.0,0.0)";
    return 0;
}
```

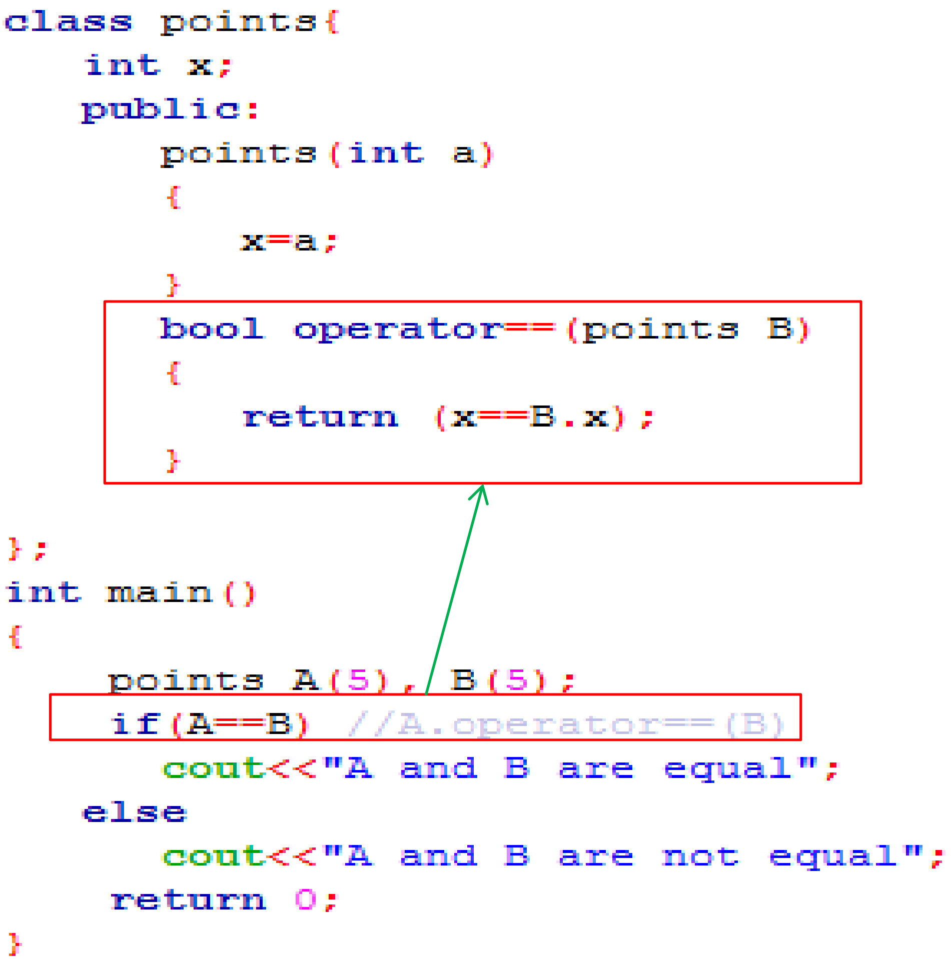
For (0.0, 0.0), the logical not operator will return true

== Operator overloading



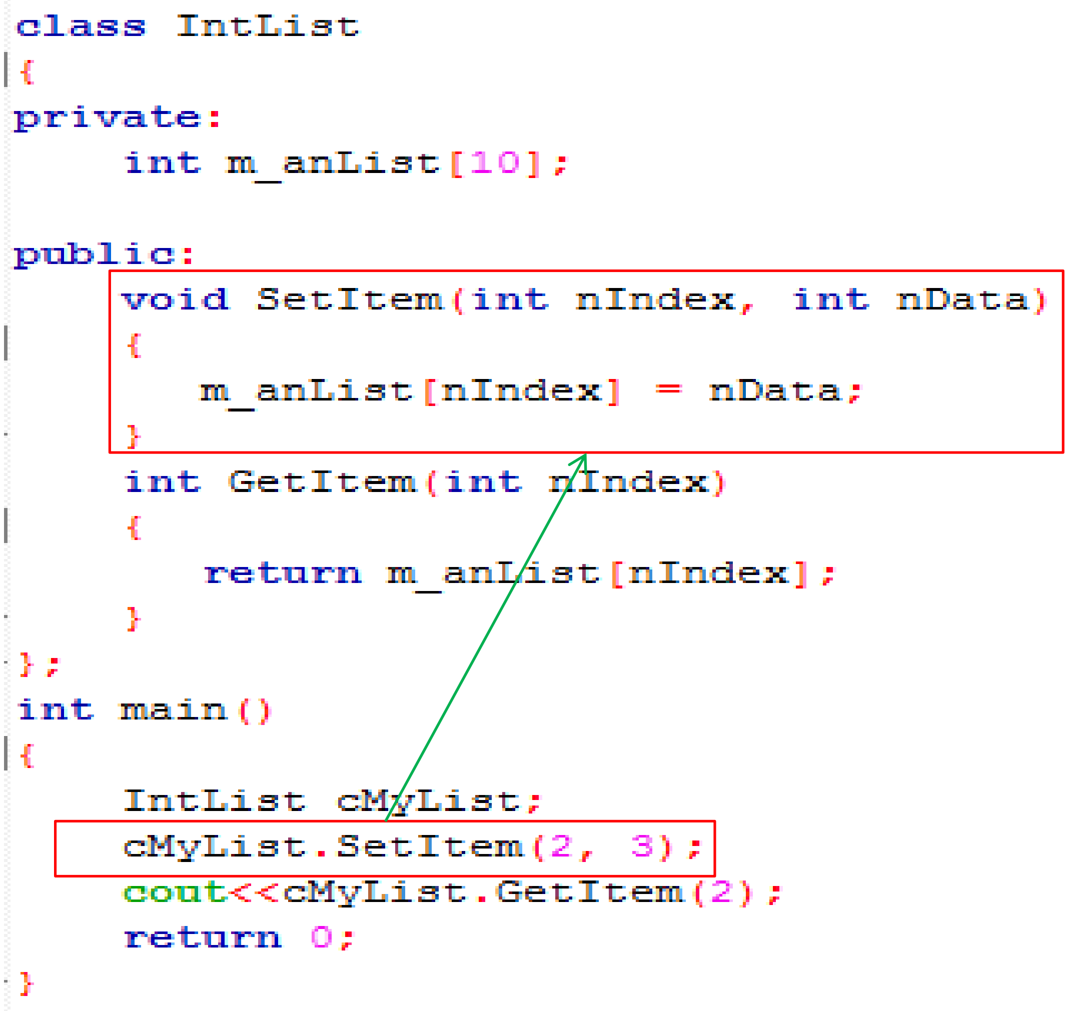
```
class points{
    int x;
    public:
        points(int a)
        {
            x=a;
        }
        bool operator==(points B)
        {
            return (x==B.x);
        }


};
int main()
{
      points A(5), B(5);
      if(A==B) //A.operator==(B)
         cout<<"A and B are equal";
    else
         cout<<"A and B are not equal";
      return 0;
}
```

== Operator overloading



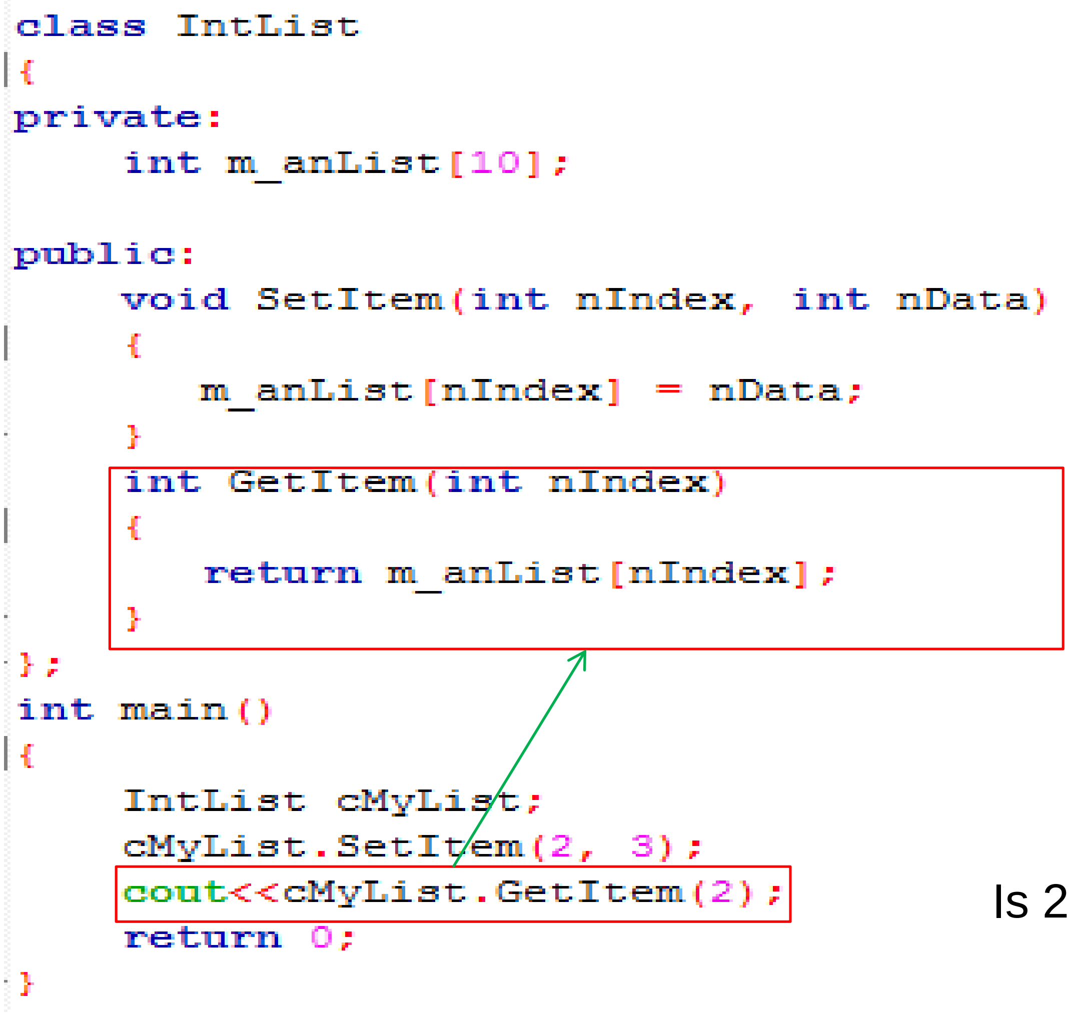
```
class points{
    int x;
    public:
        points(int a)
        {
            x=a;
        }
        bool operator==(points B)
        {
            return (x==B.x);
        }

};
int main()
{
     points A(5), B(5);
     if(A==B)  //A.operator==(B)
        cout<<"A and B are equal";
    else
        cout<<"A and B are not equal";
     return 0;
}
```

Overloading the subscript operator



```
class IntList
{
private:
    int m_anList[10];

public:
    void SetItem(int nIndex, int nData)
    {
        m_anList[nIndex] = nData;
    }
    int GetItem(int nIndex)
    {
        return m_anList[nIndex];
    }
};
int main()
{
    IntList cMyList;
    cMyList.SetItem(2, 3);
    cout<<cMyList.GetItem(2);
    return 0;
}
```

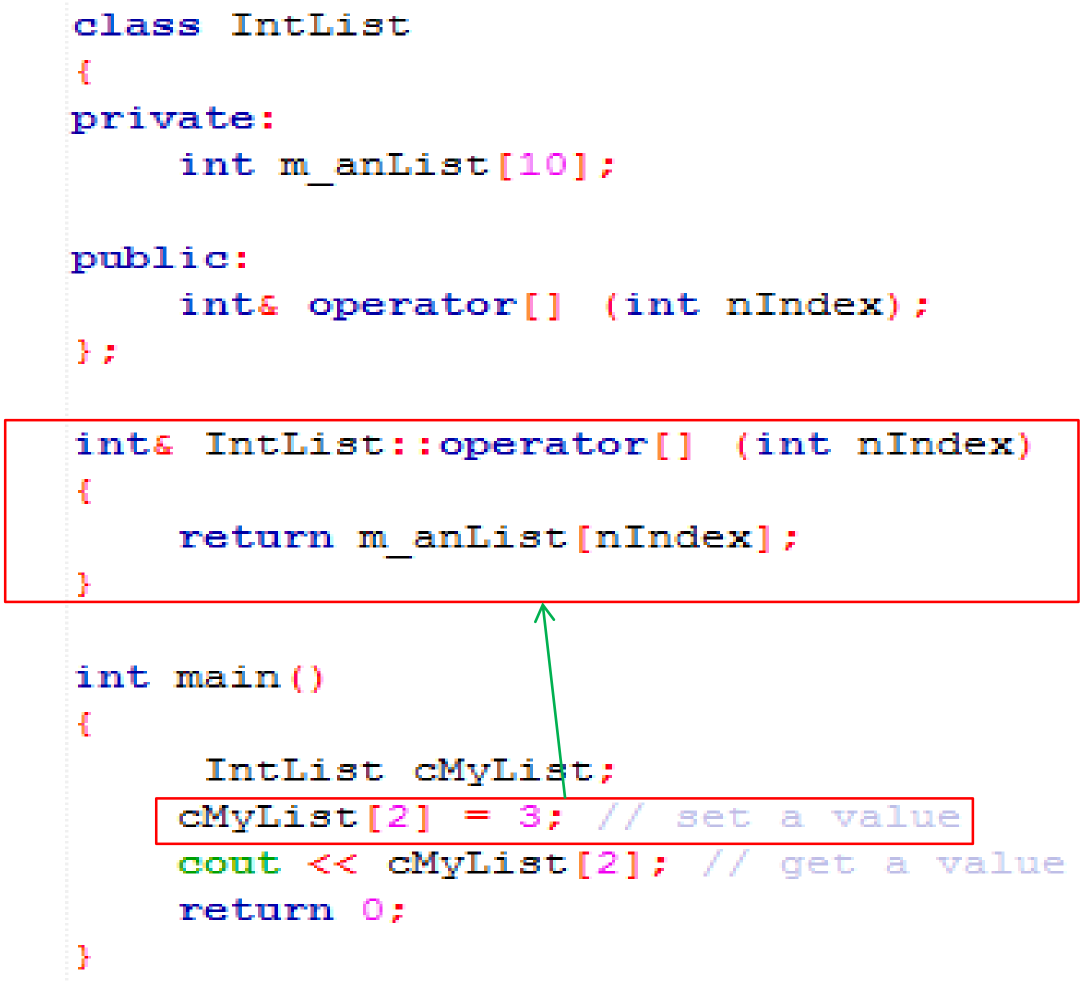
Overloading the subscript operator



```
class IntList
{
private:
    int m_anList[10];

public:
    void SetItem(int nIndex, int nData)
    {
        m_anList[nIndex] = nData;
    }
    int GetItem(int nIndex)
    {
        return m_anList[nIndex];
    }
};
int main()
{
    IntList cMyList;
    cMyList.SetItem(2, 3);
    cout<<cMyList.GetItem(2);
    return 0;
}
```

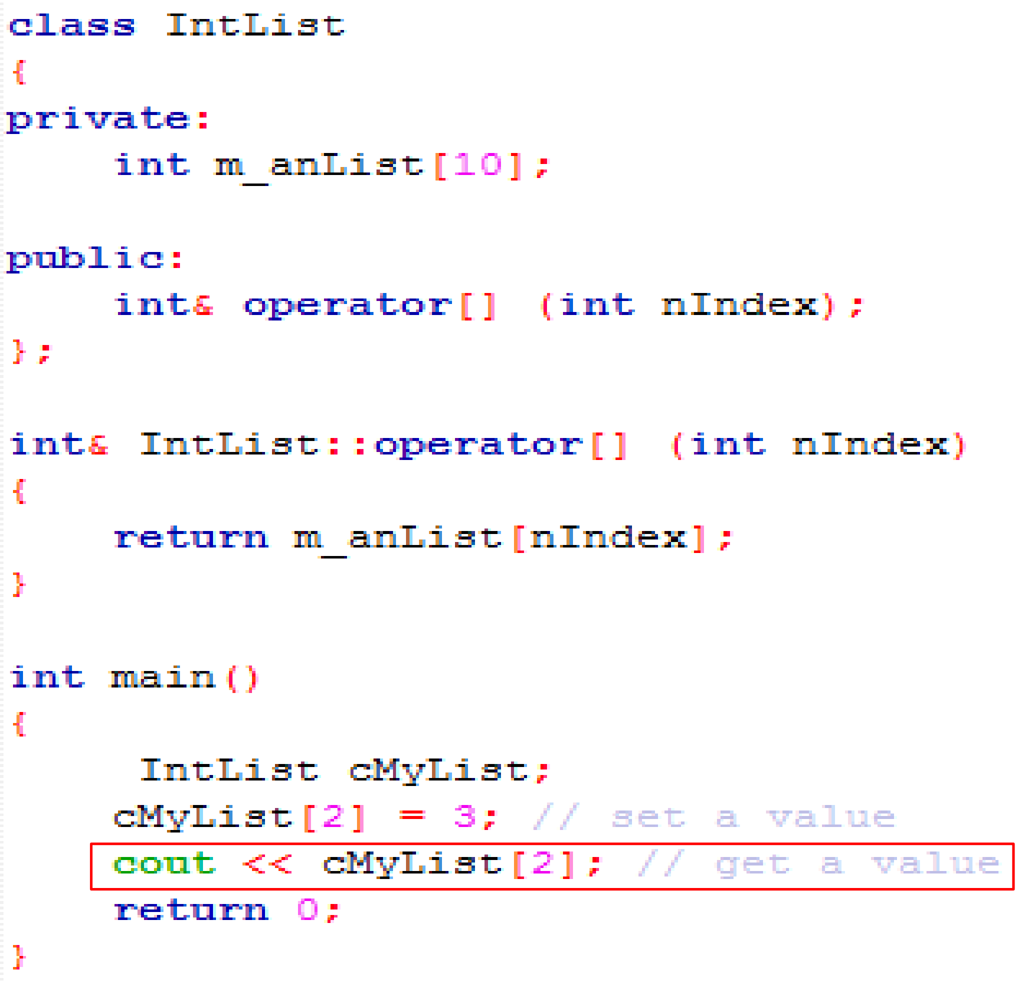
Is 2 index or 3?

Overloading the subscript operator



```
class IntList
{
private:
    int m_anList[10];

public:
    int& operator[] (int nIndex);
};

int& IntList::operator[] (int nIndex)
{
    return m_anList[nIndex];
}

int main()
{
     IntList cMyList;
    cMyList[2] = 3; // set a value
    cout << cMyList[2]; // get a value
    return 0;
}
```

Overloading the subscript operator



```
class IntList
{
private:
    int m_anList[10];

public:
    int& operator[] (int nIndex);
};

int& IntList::operator[] (int nIndex)
{
    return m_anList[nIndex];
}

int main()
{
     IntList cMyList;
    cMyList[2] = 3; // set a value
    cout << cMyList[2]; // get a value
    return 0;
}
```

Why operator[] returns a reference

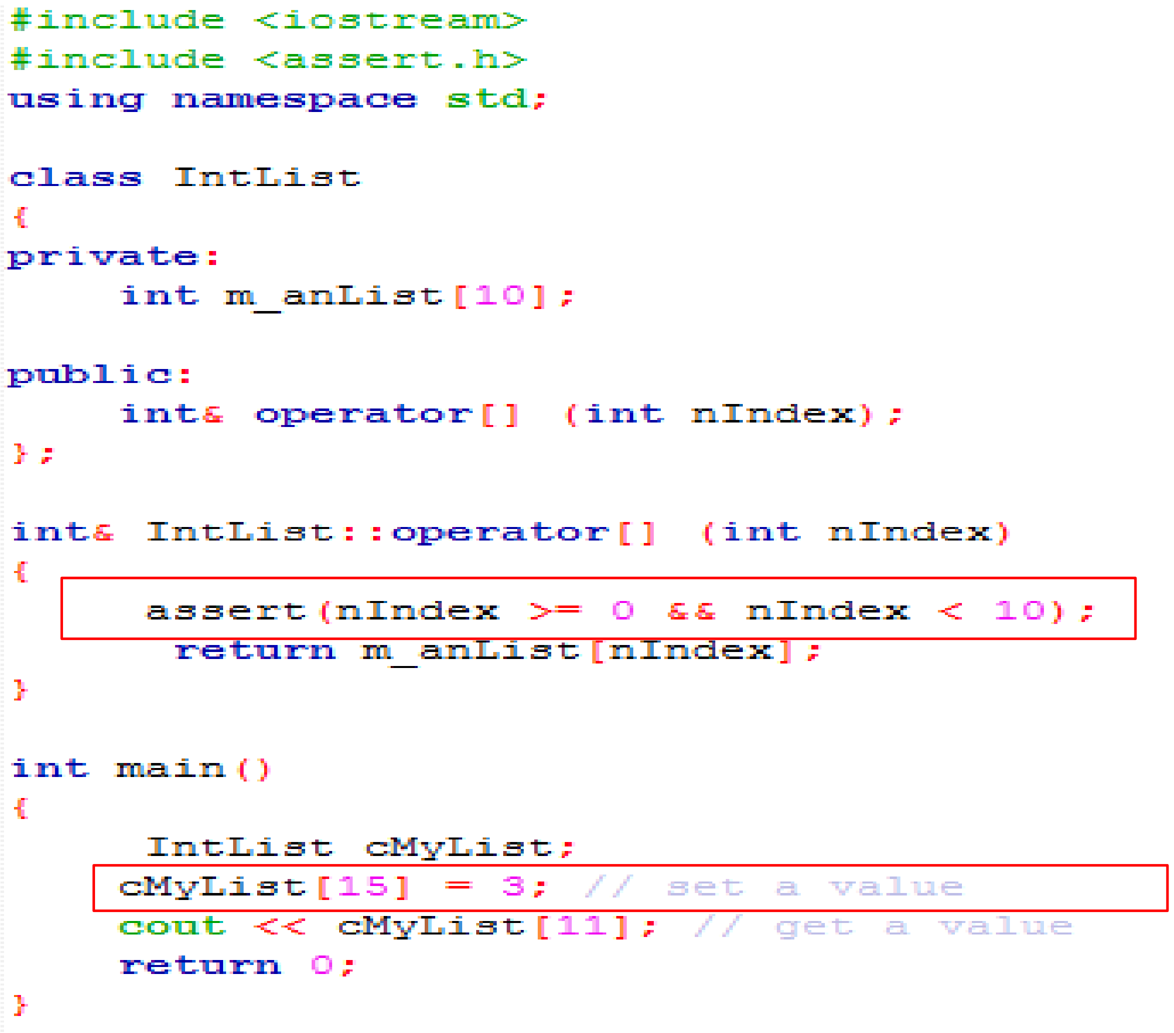
- Let’s take a closer look at how cMyList[2] = 3 evaluates. Because the subscript operator has a higher precedence than the assignment operator, cMyList[2] evaluates first.
- cMyList[2] calls operator[], which we’ve defined to return a reference to cMyList.m_anList[2].
- Because operator[] is returning a reference, it returns the actualcMyList.m_anList[2] array element.
- Our partially evaluated expression becomes cMyList.m_anList[2] = 3, which is a straightforward integer assignment.

Advantage of [ ] operator overloading

- The compiler will not complain about the following code

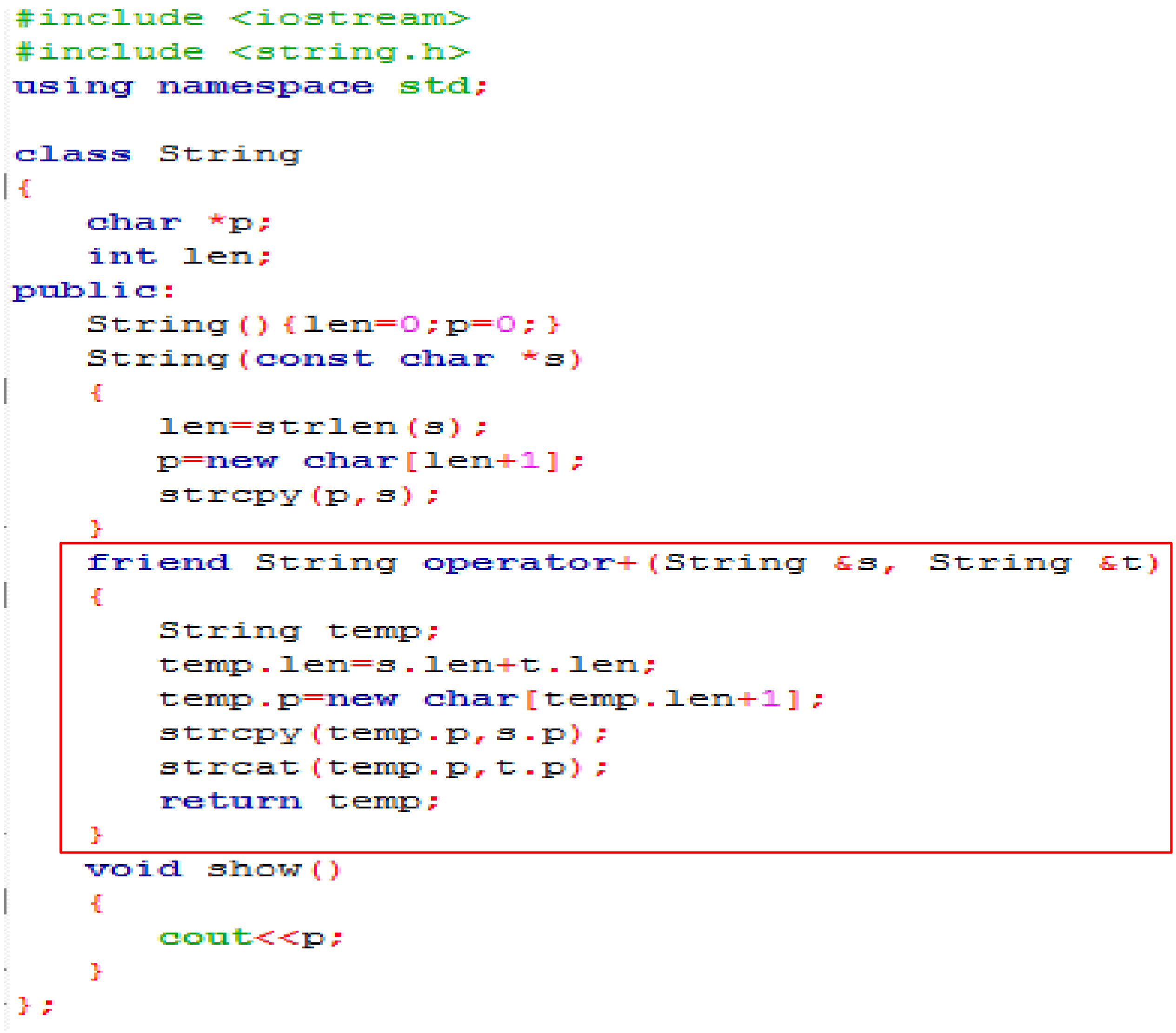
```
int anArray[5];
anArray[7] = 3; // index 7 is out of bounds!
```

Advantage of [ ] operator overloading

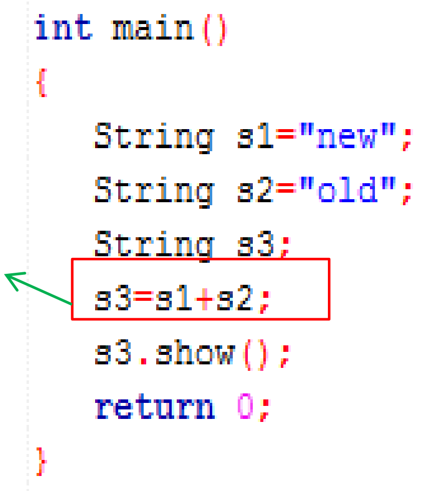


```
#include <iostream>
#include <assert.h>
using namespace std;

class IntList
{
private:
        int m_anList[10];

public:
        int& operator[] (int nIndex);
};

int& IntList::operator[] (int nIndex)
{
         assert(nIndex >= 0 && nIndex < 10);
           return m_anList[nIndex];
}

int main()
{
         IntList cMyList;
        cMyList[15] = 3; // set a value
        cout << cMyList[11]; // get a value
        return 0;
}
```

String operation



```
#include <iostream>
#include <string.h>
using namespace std;

class String
{
    char *p;
    int len;
public:
    String(){len=0;p=0;}
    String(const char *s)
    {
        len=strlen(s);
        p=new char[len+1];
        strcpy(p,s);
    }
    friend String operator+(String &s, String &t)
    {
        String temp;
        temp.len=s.len+t.len;
        temp.p=new char[temp.len+1];
        strcpy(temp.p,s.p);
        strcat(temp.p,t.p);
        return temp;
    }
    void show()
    {
        cout<<p;
    }
};
```

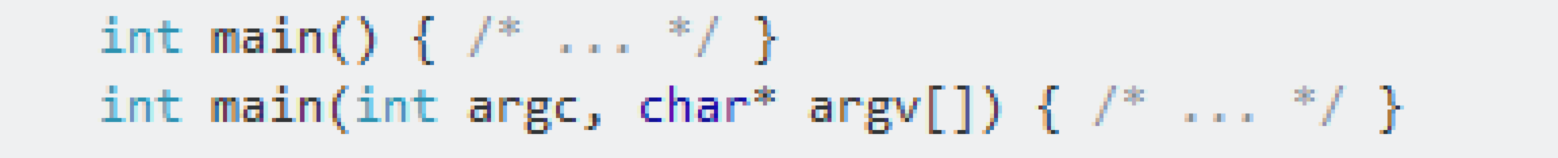


```
int main()
{
    String s1="new";
    String s2="old";
    String s3;
    s3=s1+s2;
    s3.show();
    return 0;
}
```

What about <=

Is main() overloaded in C++?

- §3.6.1/2 (C++03) says
- An implementation shall not predefine the main function.
- This function shall not be overloaded. It shall have a return type of type int, but otherwise its type is implementation-defined.
- All implementations shall allow both of the following definitions of main:



```
int main() { /* ... */ }
int main(int argc, char* argv[]) { /* ... */ }
```

Is main() overloaded in C++?

- You can use either of them.
- Both are standard compliant.
- Also, since char *argv[] is equivalent to char **argv, replacing char *argv[] with char **argv doesn't make any difference.

Is main() overloaded in C++?

- Both versions cannot co-exist at the same time. One program can have exactly one main function. Which one, depends on your choice.
- If you want to process command-line argument, then you've to choose the second version, or else first version is enough. Also note that if you use second version, and don't pass any command line argument, then there is no harm in it. It will not cause any error.
- You just have to interpret argc and argv accordingly, and based on their value, you've to write the logic and the flow of your program.

Is main() overloaded in C++?

- Don't forget that main is not usually the first thing the OS calls when executing a program. The main function is the function that is called by the run time environment. The address of the first instruction to execute is usually declared in some meta data, usually at the start if the executable file. None of the above contradicts the C/C++ standard as far as I can tell, as long as there is only one, which makes sense since the OS wouldn't know which to call if there were more than one. Checking there is only one is not done in the compiler, it is done in the linker.

References

- http://www.learncpp.com
- Object oriented programming with c++ -- E Balagurusamy
- http://stackoverflow.com

OPERATOR OVERLOADING

Animesh Kumar Paul

Rules For Overloading Operators

- Only existing operators can be overloaded. New operators cannot be created.
- The overloaded operator must have at least one operand that is of user-defined type.
- We cannot change the basic meaning of an operator.
- Overloaded operators follow the syntax rules of the original operators.

Rules For Overloading Operators continue…

- The following operators that cannot be overloaded:

Size of	Size of operator
.	Membership operator
.*	Pointer-to-member operator
: :	Scope resolution operator
? ;	Conditional operator

Rules For Overloading Operators continue…

- The following operators can be over loaded with the use of member functions and not by the use of friend functions:
  - Assignment operator =
  - Function call operator( )
  - Subscripting operator [ ]
  - Class member access operator ->
- Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument.

Rules For Overloading Operators

continue…

- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

Case Study: An Array class

- Implement an Array class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with << and >>
  - Array comparisons with == and !=

Case Study: `Array` Class

- Example: Implement an `Array` class with
  - Range checking
  - Array assignment
  - Arrays that know their own size
  - Outputting/inputting entire arrays with << and >>
  - Array comparisons with == and `!=`

11.8 Case Study: `Array` Class (Cont.)

- Copy constructor
  - Used whenever copy of object is needed:
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another of same type
      - `Array newArray( oldArray );` or `Array newArray = oldArray` (both are identical)
        - `newArray` is a copy of `oldArray`
  - Prototype for class `Array`
    - `Array( const Array & );`
    - Must take reference
      - Otherwise, the argument will be passed by value…
      - Which tries to make copy by calling copy constructor…
        - Infinite loop

`Array.h`

(1 of 2)

```cpp
// Fig. 11.6: Array.h
// Array class for storing arrays of integers.
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using std::ostream;
using std::istream;

class Array
{
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 ); // default constructor
   Array( const Array & ); // copy constructor
   ~Array(); // destructor
   int getSize() const; // return size

   const Array &operator=( const Array & ); // assignment operator
   bool operator==( const Array & ) const; // equality operator

   // inequality operator; returns opposite of == operator
   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right ); // invokes Array::operator==
   } // end function operator!=
```

Most operators overloaded as member functions (except << and >>, which must be global functions)

Prototype for copy constructor

!= operator simply returns opposite of == operator – only need to define the == operator

```

   // subscript operator for non-const objects returns modifiable lvalue
   int &operator[]( int );

   // subscript operator for const objects returns rvalue
   int operator[]( int ) const;
private:
   int size; // pointer-based array size
   int *ptr; // pointer to first element of pointer-based ar
}; // end class Array

#endif
```

`Array.h`

Operators for accessing specific elements of **`Array`** object

```cpp
// Fig 11.7: Array.cpp
// Member-function definitions for class Array
#include <iostream>
using std::cerr;
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // exit function prototype
using std::exit;

#include "Array.h" // Array class definition

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor
```

Array.cpp

(1 of 6)

```

// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
   delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
   return size; // number of elements in Array
} // end function getSize
```

We must declare a new integer array so the objects do not point to the same memory

Outline

Array.cpp

(2 of 6)

Array.cpp

(3 of 6)

```cpp

// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) // avoid self-assignment
   {
      // for Arrays of different sizes, deallocate original
      // left-side array, then allocate new left-side array
      if ( size != right.size )
      {
         delete [] ptr; // release space
         size = right.size; // resize this object
         ptr = new int[ size ]; // create space for array copy
      } // end inner if

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ]; // copy array into object
   } // end outer if

   return *this; // enables x = y = z, for example
} // end function operator=
```

Want to avoid self assignment

This would be dangerous if **`this`** is the same **`Array`** as **`right`**

```cpp

// determine if two Arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false; // arrays of different number of elements

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // Array contents are not equal

   return true; // Arrays are equal
} // end function operator==

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
      cerr << "\nError: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program; subscript out of range
   } // end if

   return ptr[ subscript ]; // reference return
} // end function operator[]
```

`integers1[ 5 ]` calls `integers1.operator[]( 5 )`

Outline

Array.cpp

(4 of 6)

```cpp

// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[]( int subscript ) const
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
      cerr << "\nError: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program; subscript out of range
   } // end if

   return ptr[ subscript ]; // returns copy of this element
} // end function operator[]

// overloaded input operator for class Array;
// inputs values for entire Array
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input; // enables cin >> x >> y;
} // end function
```

Outline

Array.cpp

(5 of 6)

```cpp

// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   // output private ptr-based array
   for ( i = 0; i < a.size; i++ )
   {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;
   } // end for

   if ( i % 4 != 0 ) // end last line of output
      output << endl;

   return output; // enables cout << x << y;
} // end function operator<<
```

Outline

`Array.cpp`

(6 of 6)

fig11_08.cpp

(1 of 5)

```cpp
// Fig. 11.8: fig11_08.cpp
// Array class test program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "Array.h"

int main()
{
   Array integers1( 7 ); // seven-element Array
   Array integers2; // 10-element Array by default

   // print integers1 size and contents
   cout << "Size of Array integers1 is "
      << integers1.getSize()
      << "\nArray after initialization:\n" << integers1;

   // print integers2 size and contents
   cout << "\nSize of Array integers2 is "
      << integers2.getSize()
      << "\nArray after initialization:\n" << integers2;

   // input and print integers1 and integers2
   cout << "\nEnter 17 integers:" << endl;
   cin >> integers1 >> integers2;
```

Retrieve number of elements in **`Array`**

Use overloaded >> operator to input

```

   cout << "\nAfter input, the Arrays contain:\n"
      << "integers1:\n" << integers1
      << "integers2:\n" << integers2;

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2" << endl;

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal" << endl;

   // create Array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3( integers1 ); // invokes copy constructor

   cout << "\nSize of Array integers3 is "
      << integers3.getSize()
      << "\nArray after initialization:\n" << integers3;

   // use overloaded assignment (=) operator
   cout << "\nAssigning integers2 to integers1:" << endl;
   integers1 = integers2; // note target Array is smaller

   cout << "integers1:\n" << integers1
      << "integers2:\n" << integers2;

   // use overloaded equality (==) operator
   cout << "\nEvaluating: integers1 == integers2" << endl;
```

Use overloaded **<<** operator to output

Use overloaded **!=** operator to test for inequality

Use copy constructor

Use overloaded **=** operator to assign

```cpp

   if ( integers1 == integers2 )
      cout << "integers1 and integers2 are equal" << endl;

   // use overloaded subscript operator to create rvalue
   cout << "\nintegers1[5] is " << integers1[ 5 ];

   // use overloaded subscript operator to create lvalue
   cout << "\n\nAssigning 1000 to integers1[5]" << endl;
   integers1[ 5 ] = 1000;
   cout << "integers1:\n" << integers1;

   // attempt to use out-of-range subscript
   cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
   integers1[ 15 ] = 1000; // ERROR: out of range
   return 0;
} // end main
```

Use overloaded == operator to test for equality

`fig11_08.cpp`

(3 of 5)

Use overloaded `[]` operator to access individual integers, with range-checking

```
Size of Array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of Array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Outline

fig11_08.cpp

(4 of 5)

```
Size of Array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

Outline

fig11_08.cpp

(5 of 5)

Consideration

- The argument to a copy constructor should be a `const` reference to allow a `const` object to be copied.

Error

- Note that a copy constructor *must* receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!

Error

- If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both objects would point to the same dynamically allocated memory. The first destructor to execute would then delete the dynamically allocated memory, and the other object's `ptr` would be undefined, a situation called a dangling pointer—this would likely result in a serious run-time error (such as early program termination) when the pointer was used.

Consideration

- A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

Consideration

- It is possible to prevent one object of a class from being assigned to another. This is done by declaring the assignment operator as a `private` member of the class.

Consideration

- It is possible to prevent class objects from being copied; to do this, simply make both the overloaded assignment operator and the copy constructor of that class `private`.

Case Study: `String` Class

- Build class `String`
  - String creation, manipulation
  - Similar to class `string` in standard library
- Conversion constructor
  - Any single-argument constructor
  - Turns objects of other types into class objects
    - Example
      - `String s1( "happy" );`
        - Creates a `String` from a `char *`
- Overloading function call operator
  - Powerful (functions can take arbitrarily long and complex parameter lists)

String.h

3)

```
// Fig. 11.9: String.h
// String class definition.
#ifndef STRING_H
#define STRING_H

#include <iostream>
using std::ostream;
using std::istream;

class String
{
   friend ostream &operator<<( ostream &, const String & );
   friend istream &operator>>( istream &, String & );
public:
   String( const char * = "" ); // conversion/default constructor
   String( const String & ); // copy constructor
   ~String(); // destructor

   const String &operator=( const String & ); // assignment operator
   const String &operator+=( const String & ); // concatenation operator

   bool operator!() const; // is String empty?
   bool operator==( const String & ) const; // test s1 == s
   bool operator<( const String & ) const; // test s1 < s2

```

Conversion constructor to make a **String** from a **char ***

s1 += s2 will be interpreted as **s1.operator+=(s2)**

Can also concatenate a **String** and a **char *** because the compiler will cast the **char *** argument to a **String**

String.h

(2 of 3)

```
// test s1 != s2
bool operator!=( const String &right ) const
{
   return !( *this == right );
} // end function operator!=

// test s1 > s2
bool operator>( const String &right ) const
{
   return right < *this;
} // end function operator>

// test s1 <= s2
bool operator<=( const String &right ) const
{
   return !( right < *this );
} // end function operator <=

// test s1 >= s2
bool operator>=( const String &right ) const
{
   return !( *this < right );
} // end function operator>=
```

Overload equality and relational operators

```cpp

   char &operator[]( int ); // subscript operator (modifiable lvalue)
   char operator[]( int ) const; // subscript operator (rvalue)
   String operator()( int, int = 0 ) const; // return a substring
   int getLength() const; // return string length
private:
   int length; // string length (not counting null terminator)
   char *sPtr; // pointer to start of pointer-based string

   void setString( const char * ); // utility function
}; // end class String

#endif
```

String.h

Two overloaded subscript operators, for **`const`** and non-**`const`** objects

Overload the function call operator **`()`** to return a substring

```
// Fig. 11.10: String.cpp
// Member-function definitions for class String.
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstring> // strcpy and strcat prototypes
using std::strcmp;
using std::strcpy;
using std::strcat;

#include <cstdlib> // exit prototype
using std::exit;

#include "String.h" // String class definition

// conversion (and default) constructor converts char * to String
String::String( const char *s )
   : length( ( s != 0 ) ? strlen( s ) : 0 )
{
   cout << "Conversion (and default) constructor: " << s << endl;
   setString( s ); // call utility function
} // end String conversion constructor

```

String.cpp

(1 of 7)

```
// copy constructor
String::String( const String &copy )
   : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << endl;
   setString( copy.sPtr ); // call utility function
} // end String copy constructor

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr; // release pointer-based string memory
} // end ~String destructor

// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called" << endl;

   if ( &right != this ) // avoid self assignment
   {
      delete [] sPtr; // prevents memory leak
      length = right.length; // new String length
      setString( right.sPtr ); // call utility function
   } // end if
   else
      cout << "Attempted assignment of a String to itself" << endl;

```

Outline

String.cpp

(2 of 7)

```
   return *this; // enables cascaded assignments
} // end function operator=

// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory

   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr

   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=

// is this String empty?
bool String::operator!() const
{
   return length == 0;
} // end function operator!

// Is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;
} // end function operator==

```

String.cpp

(3 of 7)

```
// Is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;
} // end function operator<

// return reference to character in String as a modifiable lvalue
char &String::operator[]( int subscript )
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
         << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]

// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
   // test for subscript out of range
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript
           << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if
```

Outline

`String.cpp`

(4 of 7)

```cpp

   return sPtr[ subscript ]; // returns copy of this element
} // end function operator[]

// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const
{
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return ""; // converted to a String object automatically

   // determine length of substring
   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';
```

`String.cpp`

(5 of 7)

```

   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr; // delete temporary array
   return tempString; // return copy of the temporary String
} // end function operator()

// return string length
int String::getLength() const
{
   return length;
} // end function getLength

// utility function called by constructors and operator=
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory

   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
      strcpy( sPtr, string2 ); // copy literal to object
   else // if string2 is a null pointer, make this an empty string
      sPtr[ 0 ] = '\0'; // empty string
} // end function setString

// overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
   output << s.sPtr;
   return output; // enables cascading
} // end function operator<<
```

String.cpp

(6 of 7)

```cpp

// overloaded input operator
istream &operator>>( istream &input, String &s )
{
   char temp[ 100 ]; // buffer to store input
   input >> setw( 100 ) >> temp;
   s = temp; // use String class assignment operator
   return input; // enables cascading
} // end function operator>>
```

Outline

String.cpp

(7 of 7)

```
// Fig. 11.11: fig11_11.cpp
// String class test program.
#include <iostream>
using std::cout;
using std::endl;
using std::boolalpha;

#include "String.h"

int main()
{
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << boolalpha << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 )
      << "\ns2 != s1 yields " << ( s2 != s1 )
      << "\ns2 >  s1 yields " << ( s2 > s1 )
      << "\ns2 <  s1 yields " << ( s2 < s1 )
      << "\ns2 >= s1 yields " << ( s2 >= s1 )
      << "\ns2 <= s1 yields " << ( s2 <= s1 );


   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:" << endl;

```

Outline

fig11_11.cpp

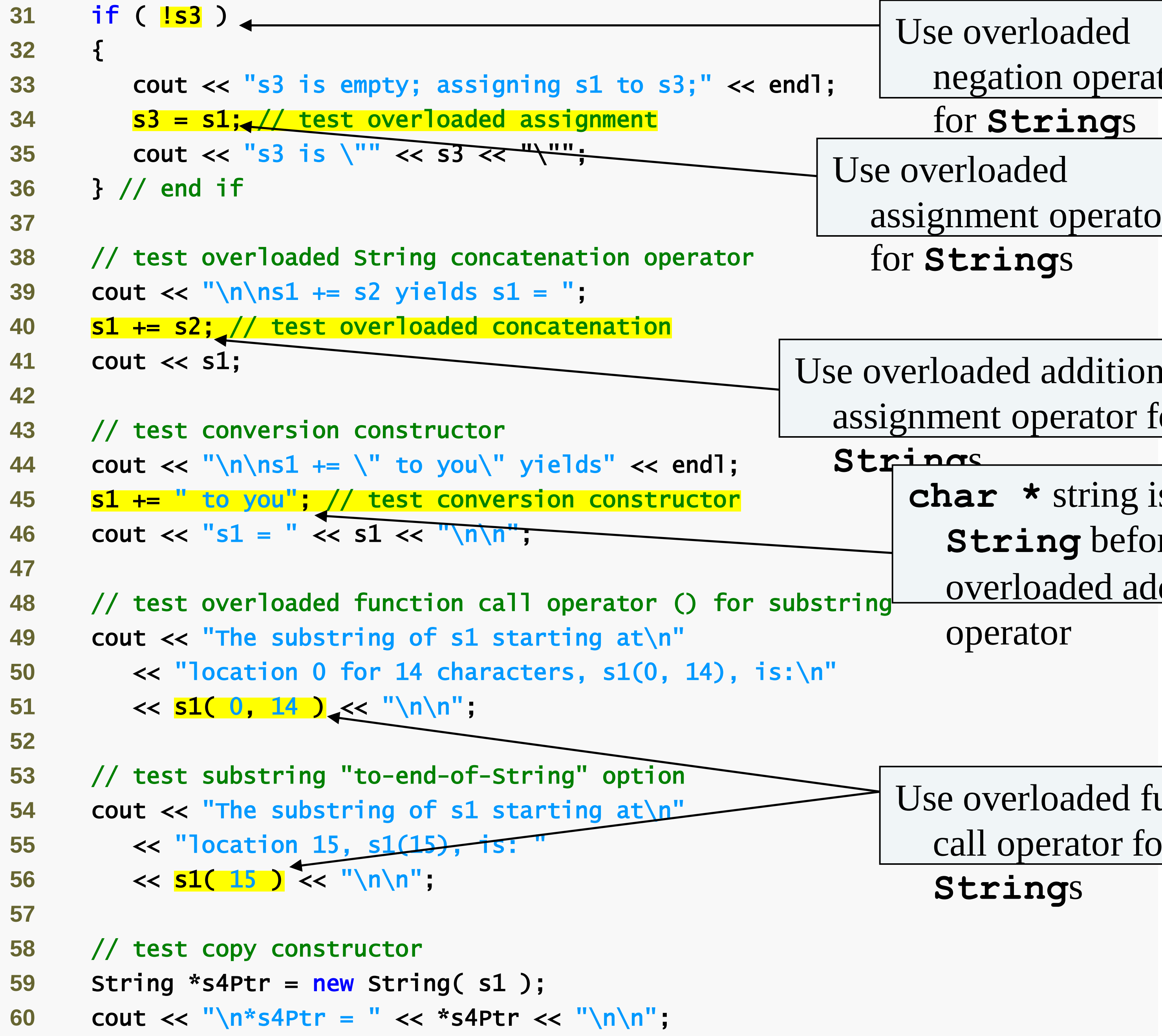
(1 of 5)

Use overloaded stream insertion operator for `String`s

Use overloaded equality and relational operators for `String`s line

g11_11.cpp

(2 of 5)



```
   if ( !s3 )
   {
      cout << "s3 is empty; assigning s1 to s3;" << endl;
      s3 = s1; // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   } // end if

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you"; // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
      << "location 0 for 14 characters, s1(0, 14), is:\n"
      << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
      << "location 15, s1(15), is: "
      << s1( 15 ) << "\n\n";

   // test copy constructor
   String *s4Ptr = new String( s1 );
   cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
```

Use overloaded negation operator for **Strings**

Use overloaded assignment operator for **Strings**

Use overloaded addition assignment operator for **Strings**

char * string is converted to a **String** before using the overloaded addition assignment operator

Use overloaded function call operator for **Strings** fig11_11.cpp

(3 of 5)

```

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr" << endl;
   *s4Ptr = *s4Ptr; // test overloaded assignment
   cout << "*s4Ptr = " << *s4Ptr << endl;

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create a modifiable lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
      << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
   s1[ 30 ] = 'd'; // ERROR: subscript out of range
   return 0;
} // end main
```

Use overloaded subscript operator for **`String`**s

Attempt to access a subscript outside of the valid range

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday
```

The constructor and destructor are called for the temporary **`String`**

(continued at top of next slide...)

Outline

`fig11_11.cpp`

(4 of 5)

(... continued from bottom of previous slide)

```
Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself


*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

fig11_11.cpp

(5 of 5)

Type Conversions

- The type conversions are automatic only when the data types involved are built-in types.

```
int m;
float x = 3.14159;
m = x; //  convert x to integer before its value is assigned
       //  to m.
```

- For user defined data types, the compiler does not support automatic type conversions.
- We must design the conversion routines by ourselves.

Type Conversions
continue…

Different situations of data conversion between incompatible types.

- Conversion from basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type.

Basic to Class Type

A constructor to build a string type object from a char * type variable.

```
string : : string(char *a)
{
   length = strlen(a);
   P = new char[length+1];
   strcpy(P,a);
}
```

The variables length and p are data members of the class string.

Basic to Class Type

continue…

```
string s1, s2;

string name1 = “IBM PC”;

string name2 = “Apple Computers”;

s1 = string(name1);

s2 = name2;
```

First converts name1 from char* type to string type and then assigns the string type value to the object s1.

First converts name2 from char* type to string type and then assigns the string type value to the object s2.

Basic to Class Type

continue…

```
class time
{      int hrs ;
       int mins ;
   public :
      ...
   time (int t)
   {
       hrs = t / 60 ;
       mins = t % 60;
   }
} ;

time T1;
int duration = 85;
T1  = duration;
```

Class To Basic Type

A constructor function do not support type conversion from a class type to a basic type.

An overloaded ***casting operator*** is used to convert a class type data to a basic type.

It is also referred to as ***conversion function.***

```
operator typename( )
{
   …
   …  ( function statements )
   …
}
```

This function converts a ***calss type*** data to ***typename.***

Class To Basic Type

continue…

```
vector : : operator double( )
{
   double sum = 0;
   for (int i=0; i < size ; i++)
      sum = sum + v[i] * v[i];
   return sqrt (sum);
}
```

This function converts a vector to the square root of the sum of squares of its components.

Class To Basic Type

continue…

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

```
vector : : operator double( )
{
   double sum = 0;
   for (int i=0; i < size ; i++)
      sum = sum + v[i] * v[i];
   return sqrt (sum);
}
```

```
double length = double(v1)
double length = v1;
```

Class To Basic Type

continue…

- Conversion functions are member functions and it is invoked with objects.
- Therefore the values used for conversion inside the function belong to the object that invoked the function.
- This means that the function does not need an argument.

One Class To Another Class Type

objX = objY ; // objects of different types

- **objX** is an object of class **X** and **objY** is an object of class **Y.**
- The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX.**
- Conversion is takes place from **class Y** to **class X.**
- **Y** is known as ***source class.***
- **X** is known as ***destination class.***

One Class To Another Class Type continue…

Conversion between objects of different classes can be carried out by either a constructor or a conversion function.

Choosing of constructor or the conversion function depends upon where we want the type-conversion function to be located in the source class or in the destination class.

One Class To Another Class Type

continue…

operator typename( )

- Converts the class object of which it is a member to typename.
- The typename may be a built-in type or a user-defined one.
- In the case of conversions between objects, typename refers to the destination class.
- When a class needs to be converted, a casting operator function can be used at the source class.
- The conversion takes place in the source class and the result is given to the destination class object.

One Class To Another Class Type

continue…

Consider a constructor function with a single argument

- Construction function will be a member of the destination class.
- The argument belongs to the source class and is passed to the destination class for conversion.
- The conversion constructor be placed in the destination class.

Data conversion

□

```
#include <iostream>
using namespace std;
class stock2 ;
class stock1
{
  int code , item ;
  float price ;
  public :
  stock1 ( int a , int b , int c )
  {
   code = a ;
   item = b ;
   price = c ;
  };
int getcode ()
  { return code; };
  int getitem ()
  { return item ; };
  int getprice ()
  { return price ; };
void disp ()
{
 cout << " code " << code << " \n " ;
 cout << " items " << item << " \n " ;
 cout << " price per item Rs. " << price <<
" \n " ;
};

  operator float ()
  {
   return ( item*price ) ;
  };
operator stock2()
{
 stock2 temp;
 temp.code =code;
 temp.value=price*items;
 return temp;
};
};
```

Data conversion

-
  ```
  class stock2
  {
    int code ;
    float val ;
    public :
    stock2 ()
    {
      code = 0; val = 0 ;
    };
    stock2( int x , float y )
    {
      code = x ; val = y ;
    };
  ```

```
  void disp ()
  {
   cout << " code " << code << " \n " ;
   cout << " total value Rs. " << val << " \n "
;
  };
  stock2( stock1 p )
  {
    code = p.getcode() ;
    val = p.getitem() * p.getprice() ;
  };
};
```

Data conversion

```
int main()
{
  stock1 i1 ( 101 , 10 ,125.0 ) ;
  stock2 i2 ;
 //stock1 to float
  float tot_val = i1;
//stock1 to stock2
  i2 = i1 ;
  cout << " Stock Details : Stock 1 type " << " \n " ;
  i1.disp ();
  cout << " Stock Value " << " - " ;
  cout << tot_val << " \n " ;
  cout << " Stock Details : Stock 2 type " << " \n " ;
  i2.disp () ;
  return 0 ;
}
```

References

- http://www.learncpp.com
- Object oriented programming with c++ -- E Balagurusamy
- http://stackoverflow.com