C/C++ Program Design

LAB 12

CONTENTS

- Copy constructor and copy assignment operator
- Learn how to define a class with a pointer member
- Returning objects
- Using pointers to objects

2 Knowledge Points

2.1 Copy Constructor and Copy Assignment Operator

2.2 Class with a pointer as its member

2.3 Returning object

2.4 Using pointers to objects

2.1 Copy Constructor and Copy Assignment Operator

2.1.1 Copy Constructor

The **copy constructors** define what happens when an object is initialized from another object of the same type. It is used during initialization.

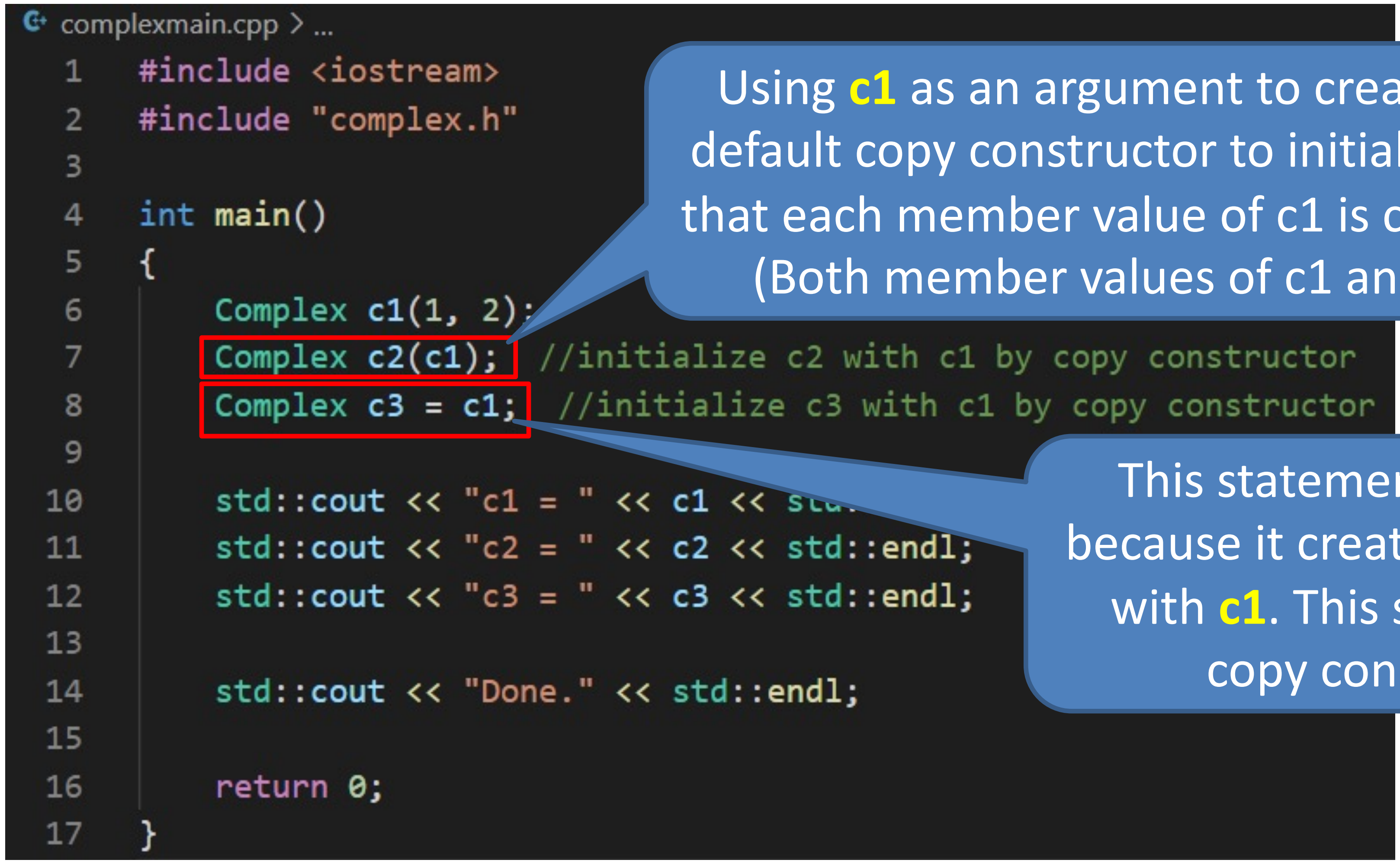
A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values. A copy constructor for a class normally has this prototype:

```
class_name (const class_name & );
```

It takes a constant reference to a class object as its argument.

When you do not define a copy constructor for a class, the compiler synthesizes one for you. Unlike the synthesized default constructor, a copy constructor is synthesized even if you define other constructors.

The default copy constructor performs a **member-to-member copy** of the nonstatic members (memberwise copying, also sometimes called ***shallow copying***). Each member is copied by value.

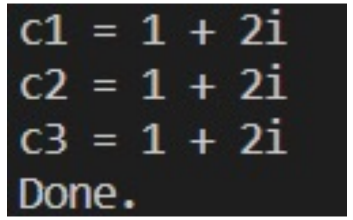


complexmain.cpp > ...

```cpp
#include <iostream>
#include "complex.h"

int main()
{
    Complex c1(1, 2);
    Complex c2(c1);  //initialize c2 with c1 by copy constructor
    Complex c3 = c1;  //initialize c3 with c1 by copy constructor

    std::cout << "c1 = " << c1 << st
    std::cout << "c2 = " << c2 << std::endl;
    std::cout << "c3 = " << c3 << std::endl;

    std::cout << "Done." << std::endl;

    return 0;
}
```

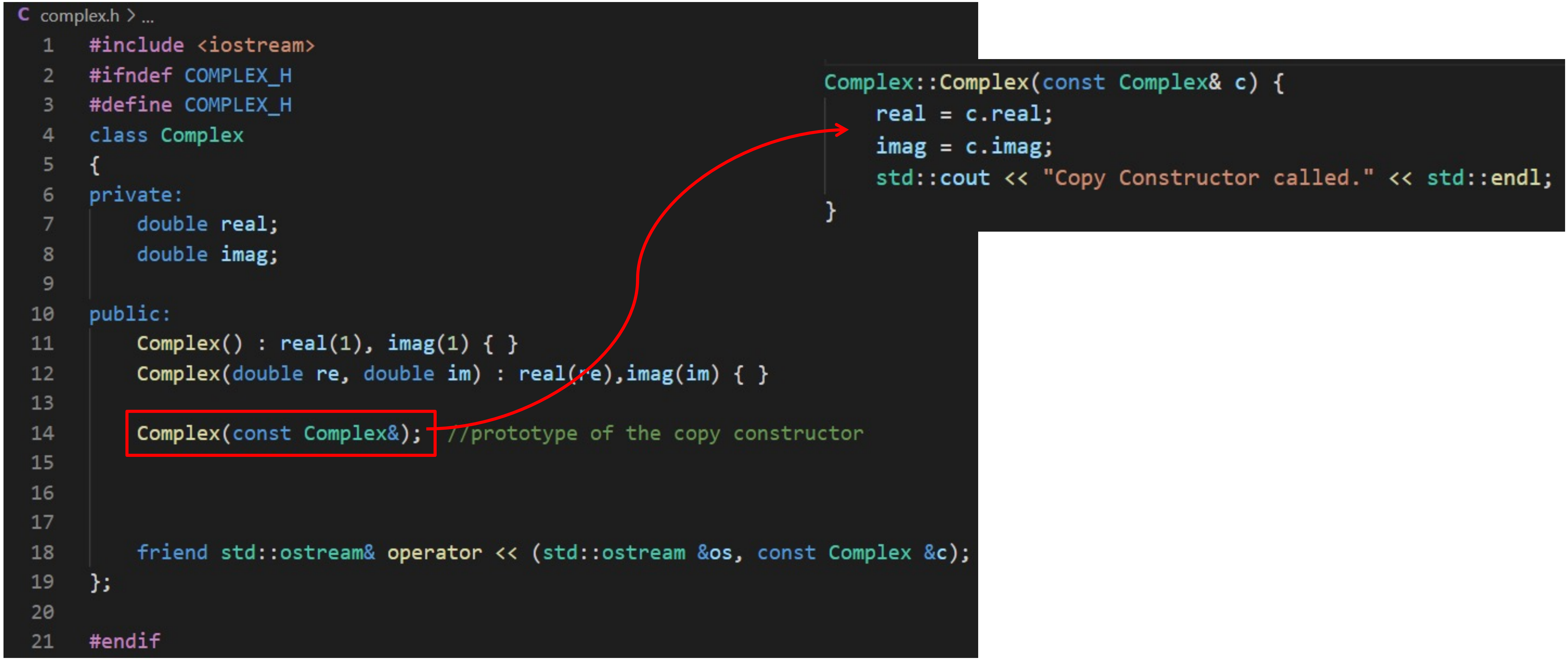
Using **c1** as an argument to create **c2** invokes the default copy constructor to initialize c2. This means that each member value of c1 is copied to that of c2 (Both member values of c1 and c2 are equal).

This statement is not an assignment statement because it creates an object **c3** and initializes the **c3** with **c1**. This statement also invokes the default copy constructor of the Complex class.



```
c1 = 1 + 2i
c2 = 1 + 2i
c3 = 1 + 2i
Done.
```

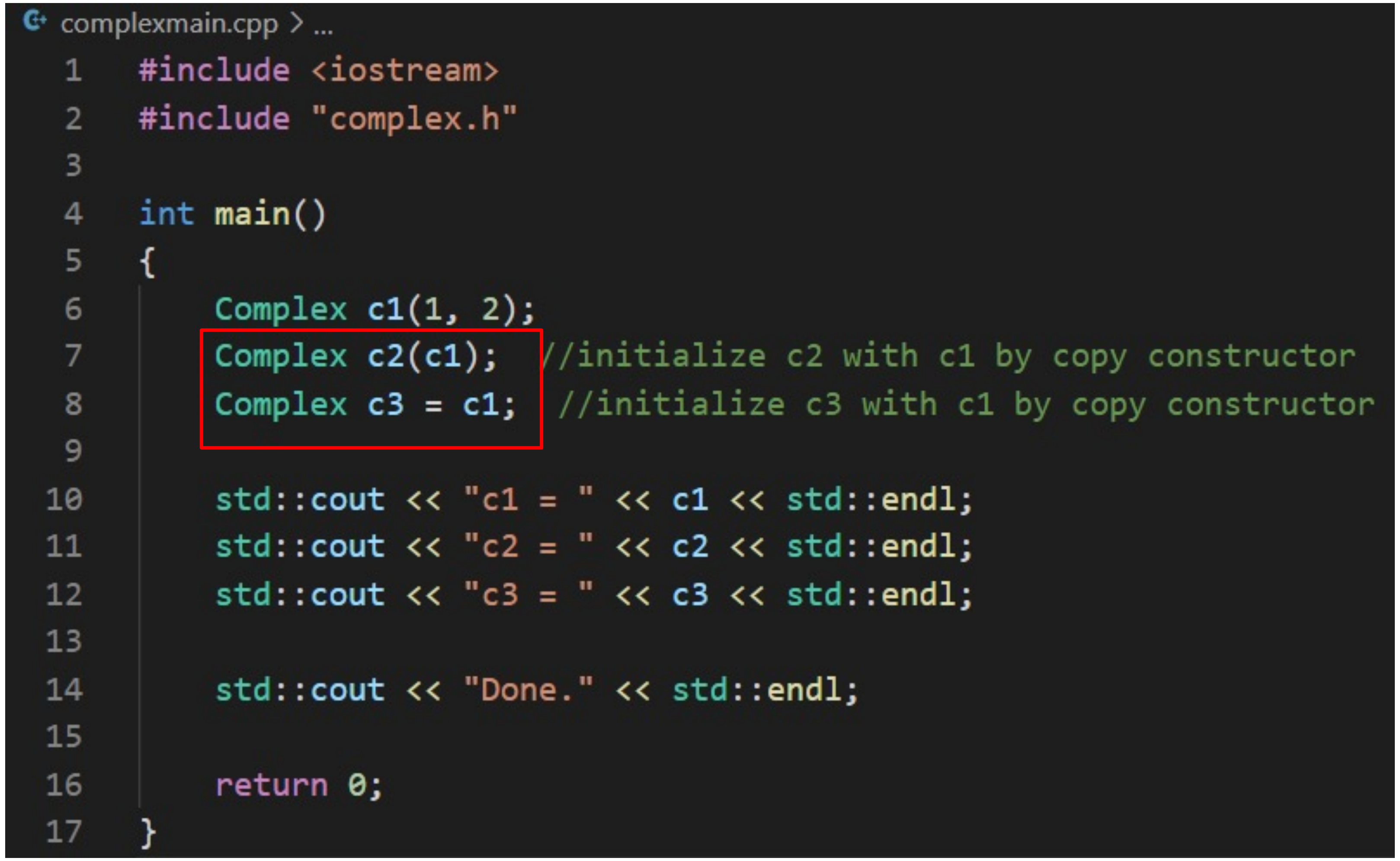
You can give an explicit copy constructor for the Complex class, this time the compiler will no longer provide the default copy constructor for you.



complex.h > ...

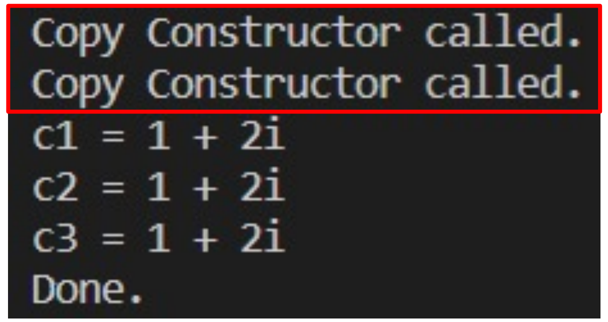
```cpp
#include <iostream>
#ifndef COMPLEX_H
#define COMPLEX_H
class Complex
{
private:
    double real;
    double imag;

public:
    Complex() : real(1), imag(1) { }
    Complex(double re, double im) : real(re),imag(im) { }

    Complex(const Complex&); //prototype of the copy constructor



    friend std::ostream& operator << (std::ostream &os, const Complex &c);
};

#endif
```

```cpp
Complex::Complex(const Complex& c) {
    real = c.real;
    imag = c.imag;
    std::cout << "Copy Constructor called." << std::endl;
}
```



complexmain.cpp > ...

```cpp
#include <iostream>
#include "complex.h"

int main()
{
    Complex c1(1, 2);
    Complex c2(c1);  //initialize c2 with c1 by copy constructor
    Complex c3 = c1;  //initialize c3 with c1 by copy constructor

    std::cout << "c1 = " << c1 << std::endl;
    std::cout << "c2 = " << c2 << std::endl;
    std::cout << "c3 = " << c3 << std::endl;

    std::cout << "Done." << std::endl;

    return 0;
}
```



```
Copy Constructor called.
Copy Constructor called.
c1 = 1 + 2i
c2 = 1 + 2i
c3 = 1 + 2i
Done.
```

A **copy constructor** is invoked whenever a new object is created and initialized to an existing object of the same kind. The following four defining declarations invoke a copy constructor:

```
Complex c2 (c1);

Complex c3 = c1;

Complex c4 = Complex(c1);

Complex *pc = new Complex(c1);
```

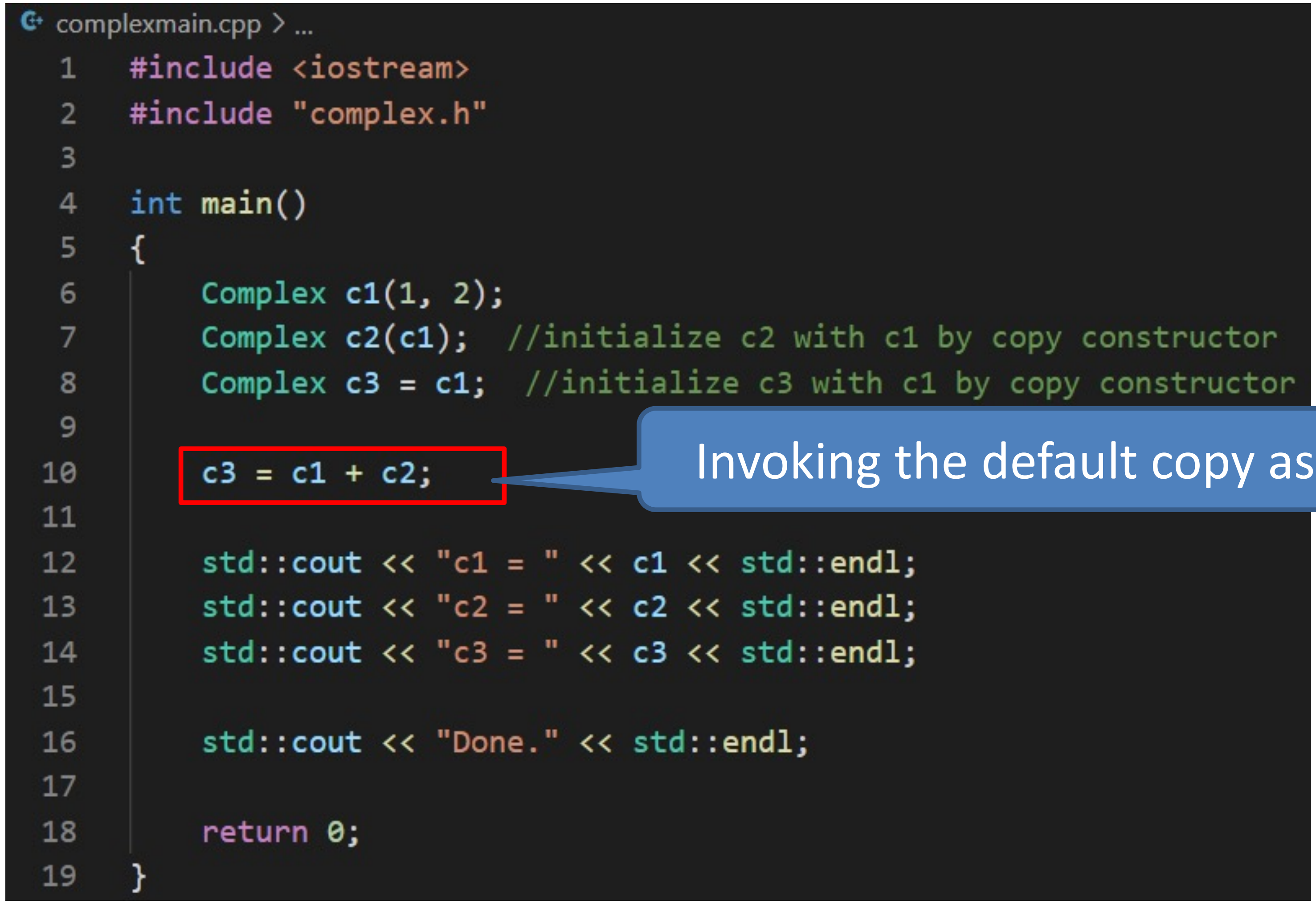
This statement initializes a anonymous object to ***c1*** and assigns the address of the new object t the ***pc*** pointer.

A copy constructor is usually called in the following situations implicitly, so it should **not be explicit**:

1. When a class object is returned by value.
2. When an object is passed to a function as an argument and is passed by value.
3. When an object is constructed from another object of the same class.
4. When a temporary object is generated by the compiler.

2.1.2 Copy Assignment Operator

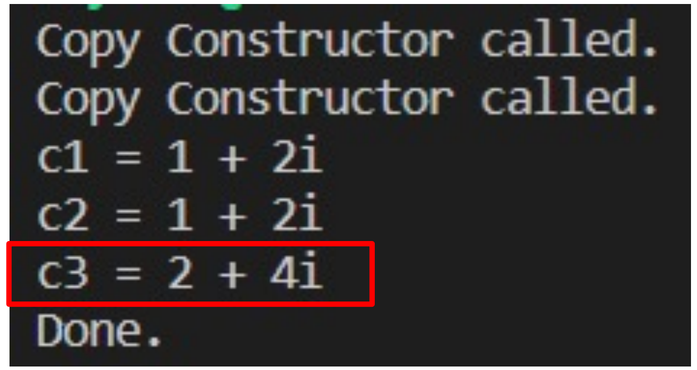
When an assignment is occurred from one object to an other, the assignment operator overloading function is invoked. The compiler synthesizes a copy-assignment operator if the class does not define its own. An implicit assignment operator performs a **member-to-member copy**.



complexmain.cpp > ...

```cpp
#include <iostream>
#include "complex.h"

int main()
{
    Complex c1(1, 2);
    Complex c2(c1);  //initialize c2 with c1 by copy constructor
    Complex c3 = c1;  //initialize c3 with c1 by copy constructor

    c3 = c1 + c2;

    std::cout << "c1 = " << c1 << std::endl;
    std::cout << "c2 = " << c2 << std::endl;
    std::cout << "c3 = " << c3 << std::endl;

    std::cout << "Done." << std::endl;

    return 0;
}
```

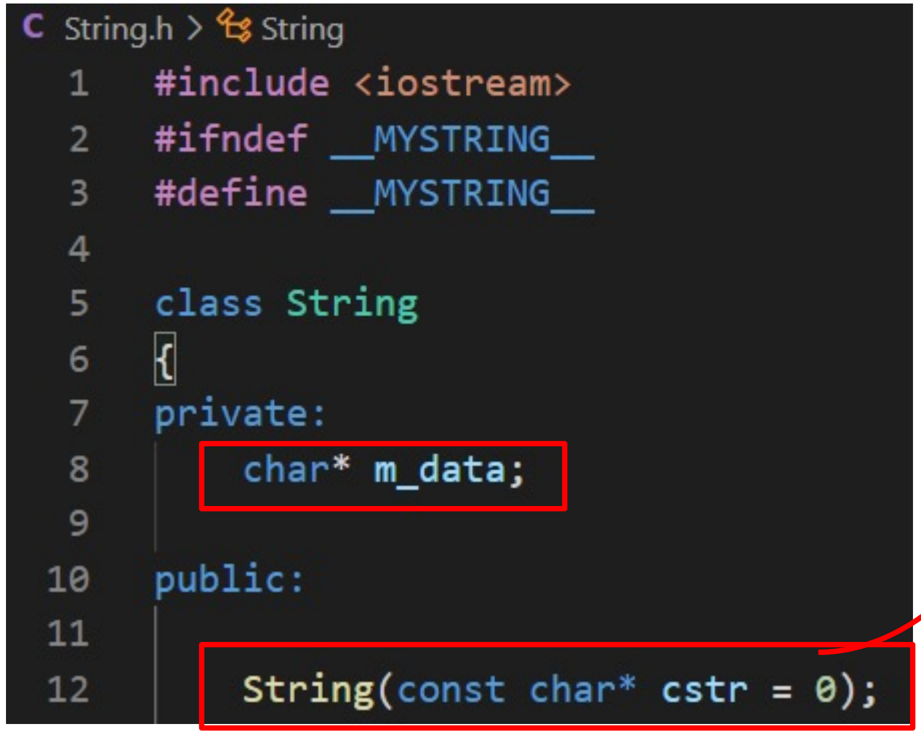
Invoking the default copy assignment operator



```
Copy Constructor called.
Copy Constructor called.
c1 = 1 + 2i
c2 = 1 + 2i
c3 = 2 + 4i
Done.
```

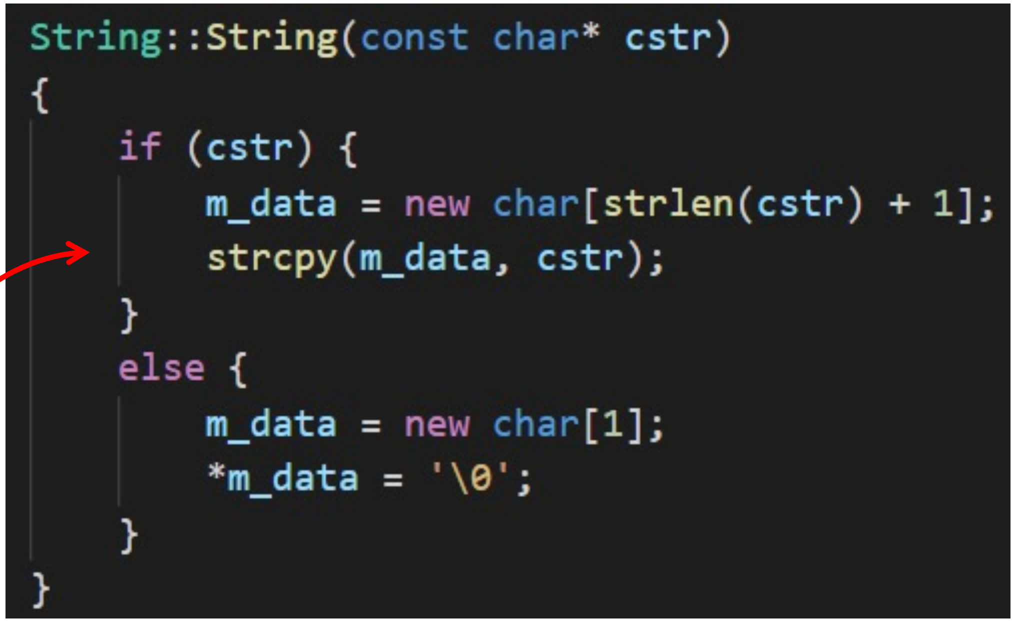
2.2 Class with a pointer as its member

If a class holds a **pointer**, how are the **constructor** and **destructor** different?



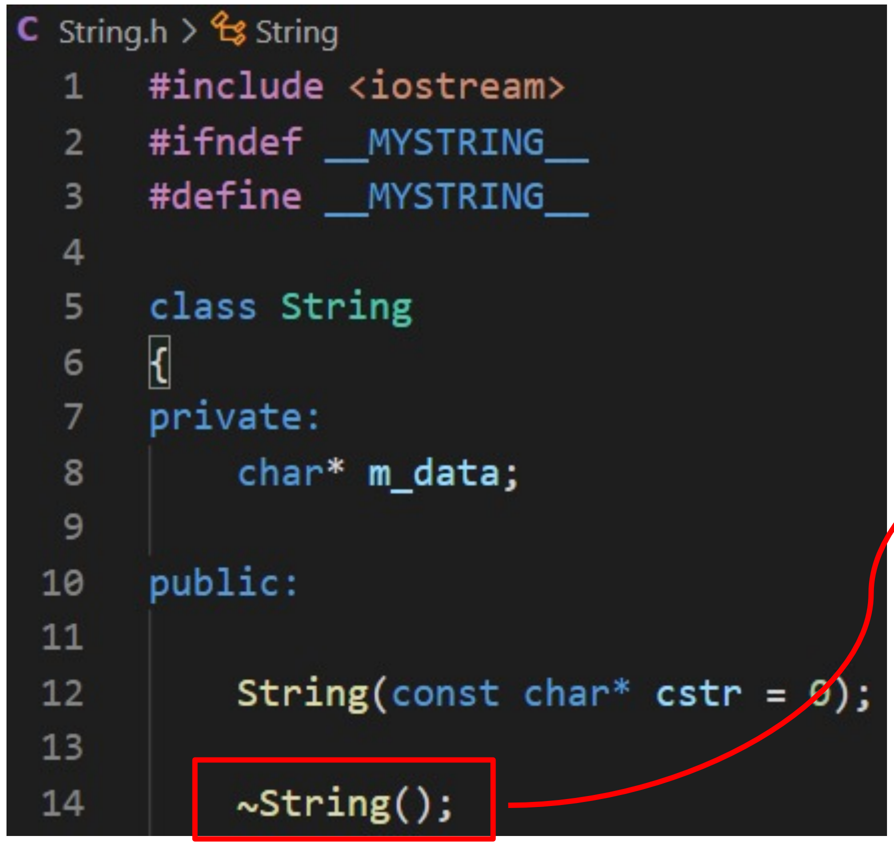
String.h > String

```
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:

    String(const char* cstr = 0);
```

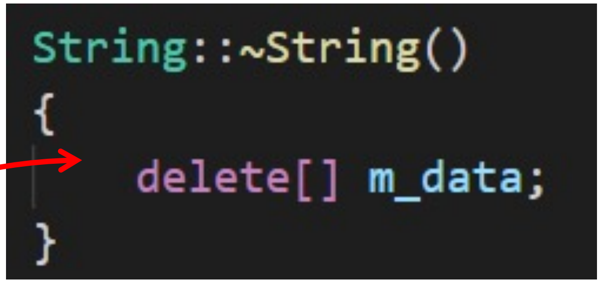


```
String::String(const char* cstr)
{
    if (cstr) {
        m_data = new char[strlen(cstr) + 1];
        strcpy(m_data, cstr);
    }
    else {
        m_data = new char[1];
        *m_data = '\0';
    }
}
```

There is a pointer-to-char member in the class definition. It should use new operator in the constructor to allocate space for the string. The constructor must **allocate enough memory** to hold the string, and then it must **copy the string** to that location.



```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:

    String(const char* cstr = 0);

    ~String();
```

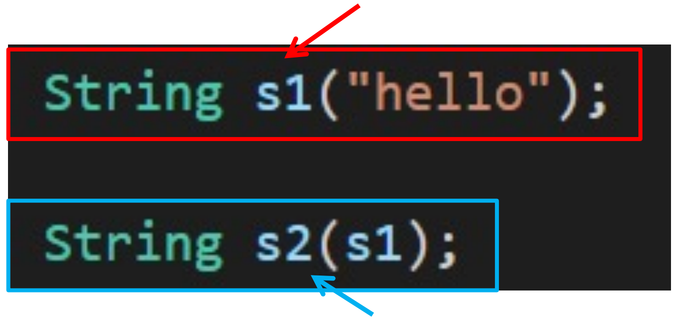


```cpp
String::~String()
{
    delete[] m_data;
}
```

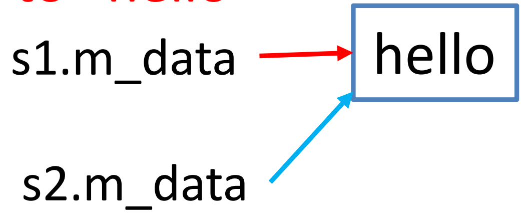
The destructor must **delete** the member which points to the memory allocated with **new**. When the String object expires, the **m_data** pointer expires. But the memory **m_data** pointed to remains allocated unless you use **delete** to free it.

If a class member holds a pointer, is the default **copy constructor** appropriate?

invoke constructor, s1.m_data points to “hello”



```
String s1("hello");

String s2(s1);
```

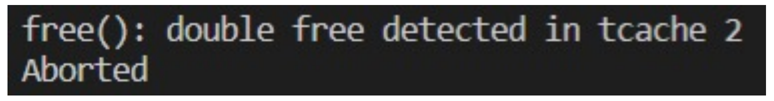


invoke copy constructor, s2.m_data points to “hello”

The default copy constructor performs a **member-to-member copy** and copies by value. This means it just copies pointer.

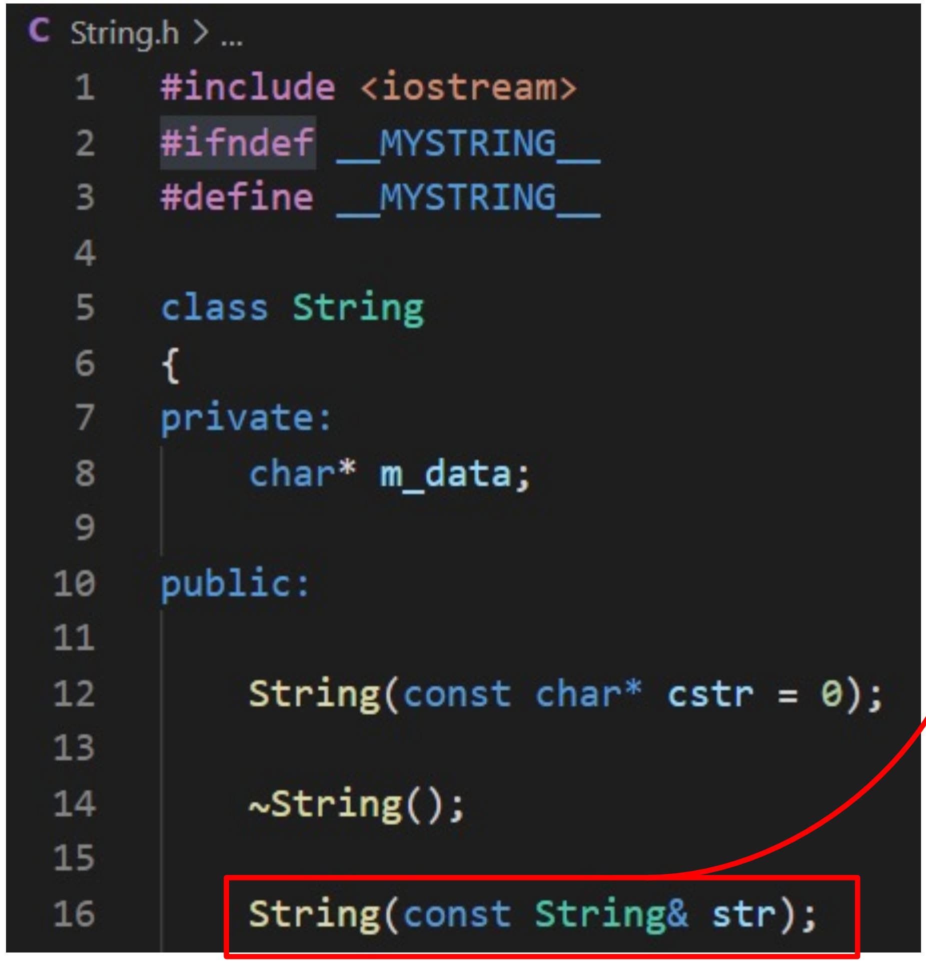
When you create s2 by s1, it invokes the default copy constructor because you don’t provide one. What the default copy constructor do is to have the two pointers points to the same string.

When object s1 is out of its scope, its destructor will be invoked, the memory where s1.m_data is pointed is free. However, when object s2 is disappear, its destructor will also be called, the memory where s2.m_data is pointed is free. These two pointers are pointed to the same memory, the same memory will be delete twice, that can cause error.



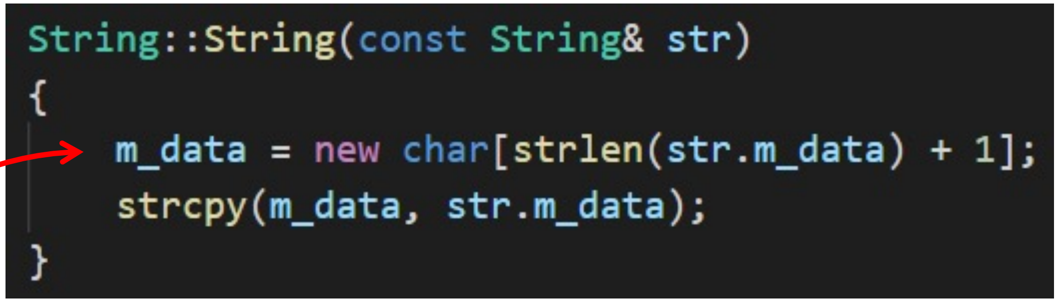
```
free(): double free detected in tcache 2
Aborted
```

You should provide an explicit copy constructor rather than default copy constructor and copy the string to the member. This is called ***deep copy***.



String.h > ...

```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:

    String(const char* cstr = 0);

    ~String();

    String(const String& str);
```

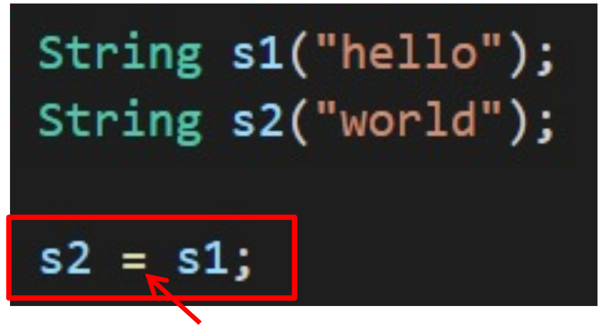


```cpp
String::String(const String& str)
{
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
}
```

What makes defining the copy constructor necessary is the fact that some class members are **new-initialized pointers to data** rather than the data themselves.

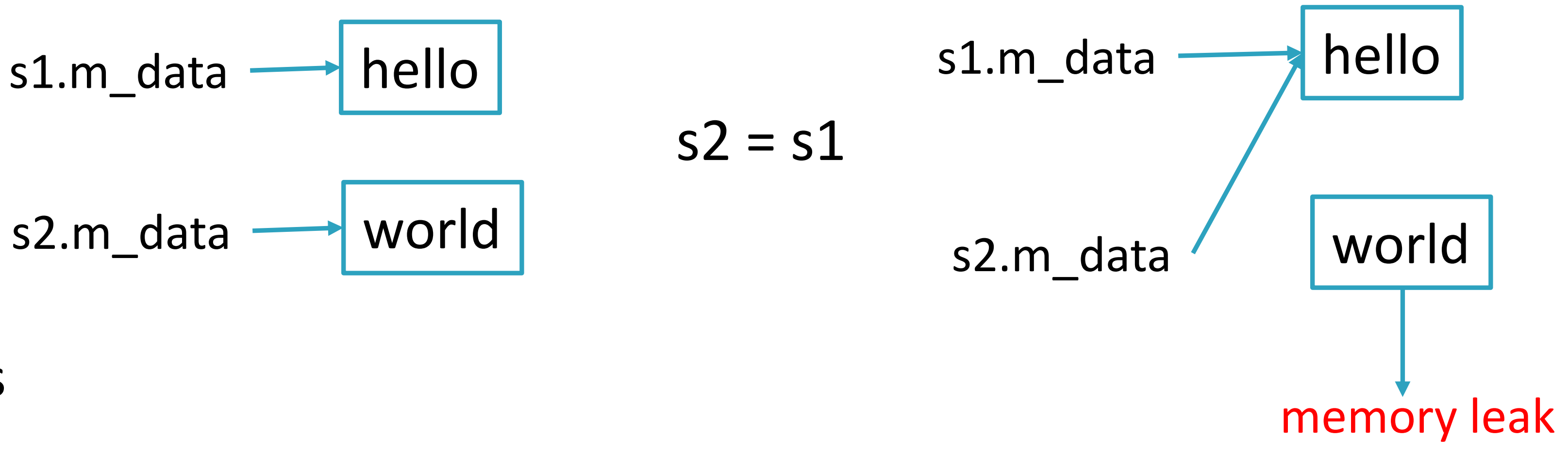
How about the default **copy assignment operator**? Is it appropriate?

The default assignment operator performs a **member-to-member copy**.



```
String s1("hello");
String s2("world");

s2 = s1;
```

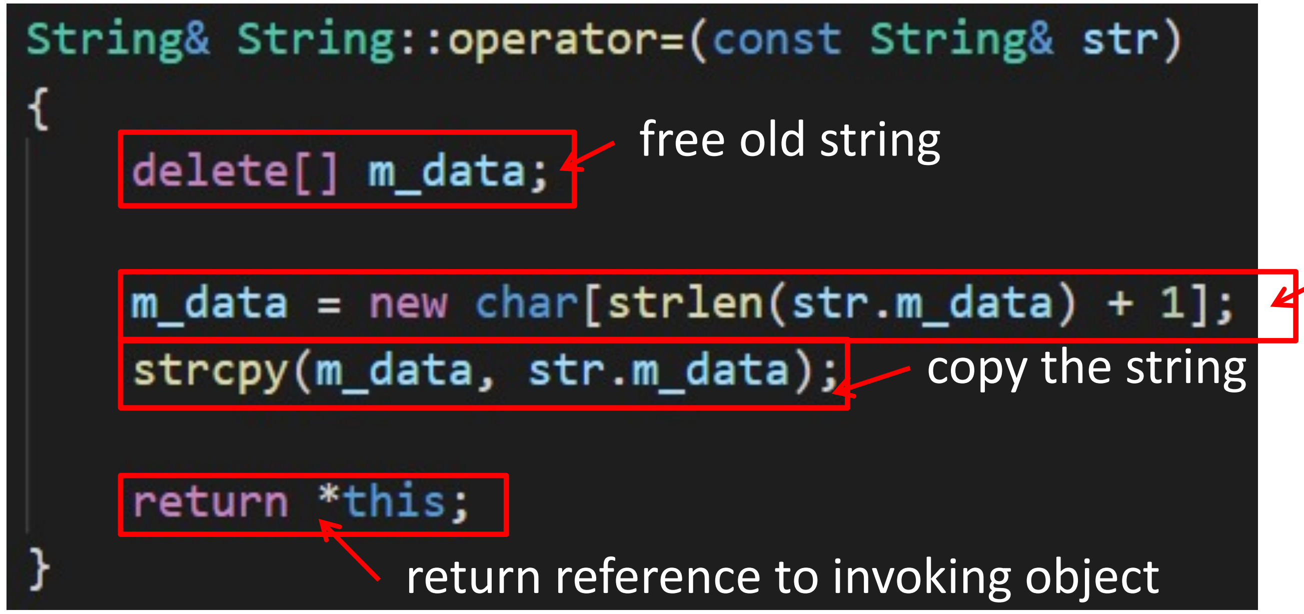
This assignment statement makes both **s1.m_data** and **s2.m_data** point to the same memory.



When release s1 and s2, the same memory will be delete twice. Besides that, the invoking default assignment operator can cause memory leak.

You should provide an explicit assignment operator definition to make a ***deep copy***. The implementations is similar to that of the copy constructor, but there are some differences:

- Check for self-assignment
- Because the target object may already refer to previously allocated data, the function should use **delete []** to free former obligations.
- **Allocate enough memory** to hold the new string, and then it must **copy the string** to that location.
- The function returns a reference to the invoking object.



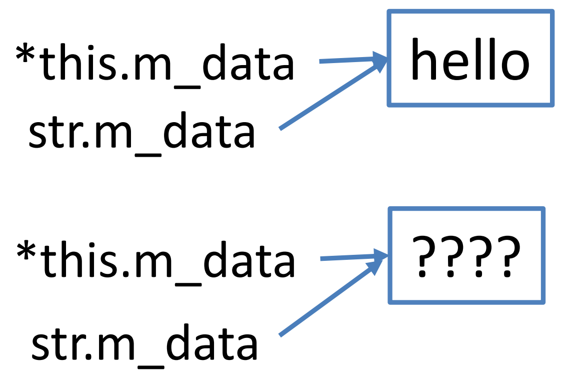
```
String& String::operator=(const String& str)
{
    delete[] m_data;

    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);

    return *this;
}
```

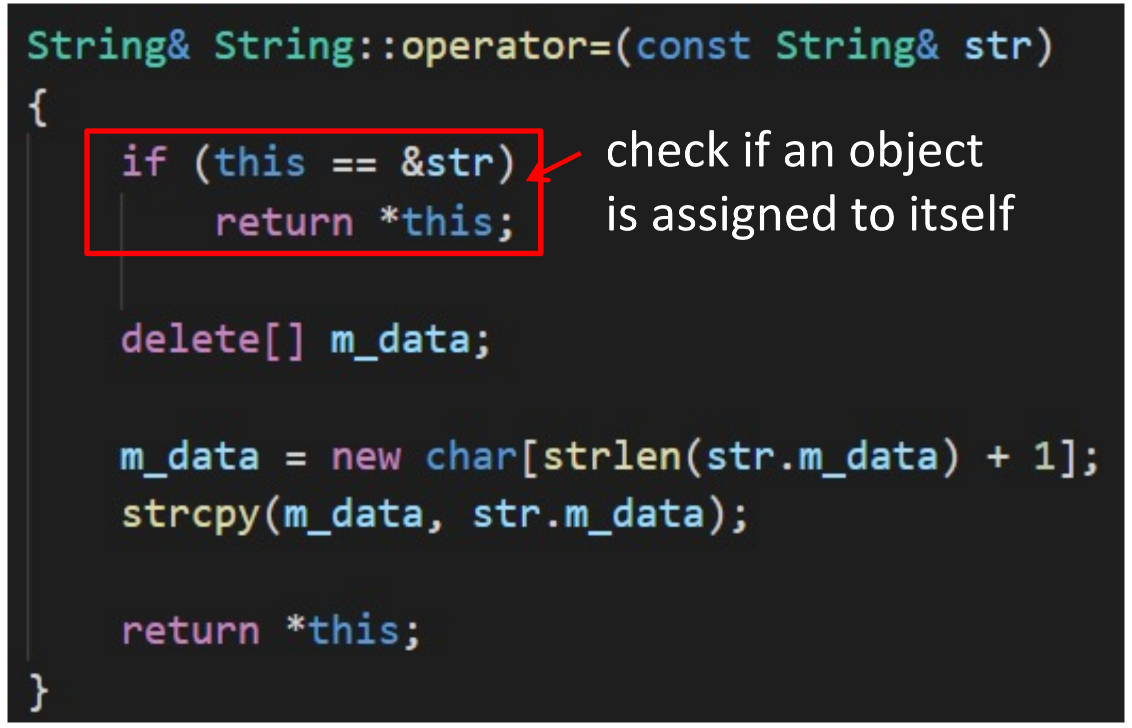
Note: we do not check self-assignment in this version.

If we do not check self-assignment, what will happen?

Suppose: we have ***s1 = s1;*** it will invoke ***operator=(const String& str)***function, ****this*** and ***str*** are the same objects.

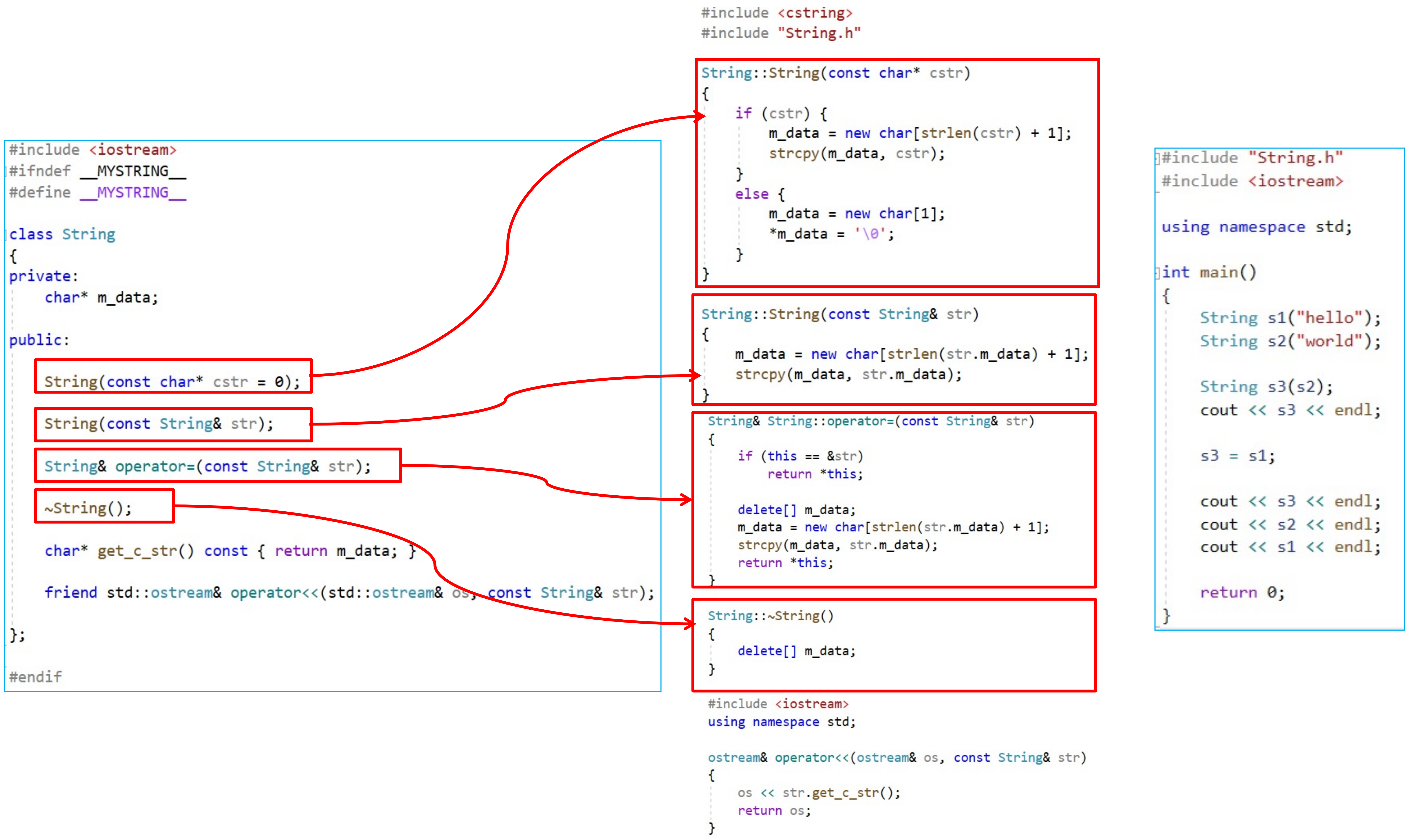
Without checking self-assignment, the statement ***delete[] m_data;*** will be executed, both ***this.m_data** and **str.m_data** point to the uncertain memory.





```
String& String::operator=(const String& str)
{
    if (this == &str)
        return *this;

    delete[] m_data;

    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);

    return *this;
}
```

Note: we do check self-assignment in this version.



```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:
    String(const char* cstr = 0);
    String(const String& str);
    String& operator=(const String& str);
    ~String();

    char* get_c_str() const { return m_data; }

    friend std::ostream& operator<<(std::ostream& os, const String& str);

};

#endif
```

```cpp
#include <cstring>
#include "String.h"

String::String(const char* cstr)
{
    if (cstr) {
        m_data = new char[strlen(cstr) + 1];
        strcpy(m_data, cstr);
    }
    else {
        m_data = new char[1];
        *m_data = '\0';
    }
}

String::String(const String& str)
{
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
}

String& String::operator=(const String& str)
{
    if (this == &str)
        return *this;

    delete[] m_data;
    m_data = new char[strlen(str.m_data) + 1];
    strcpy(m_data, str.m_data);
    return *this;
}

String::~String()
{
    delete[] m_data;
}

#include <iostream>
using namespace std;

ostream& operator<<(ostream& os, const String& str)
{
    os << str.get_c_str();
    return os;
}
```

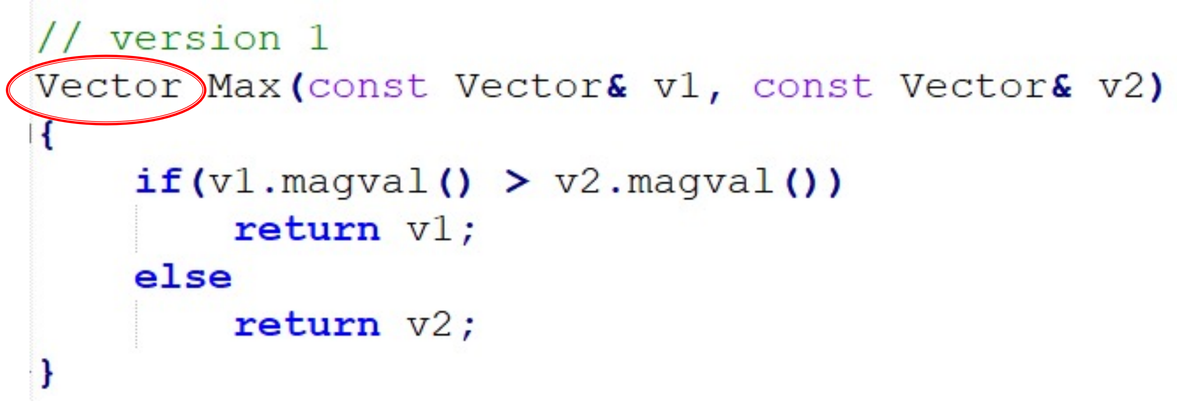
```cpp
#include "String.h"
#include <iostream>

using namespace std;

int main()
{
    String s1("hello");
    String s2("world");

    String s3(s2);
    cout << s3 << endl;

    s3 = s1;

    cout << s3 << endl;
    cout << s2 << endl;
    cout << s1 << endl;

    return 0;
}
```

2.3 Returning object

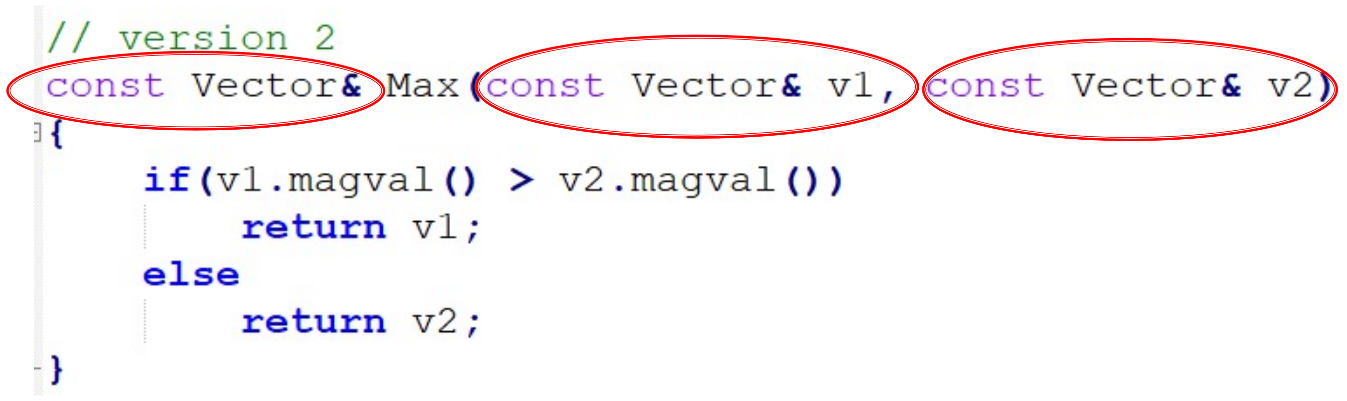
When a member function or standard function returns an object, you have choices. The function could return a **reference to an object**, a **constant reference to an object**, an **object**, or a **constant object**.

2.3.1. Returning a reference to a const object

For example, suppose you wanted to write a function Max() that returned the larger of two ***Vector*** object.



```
// version 1
Vector Max(const Vector& v1, const Vector& v2)
{
    if(v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```



```
// version 2
const Vector& Max(const Vector& v1, const Vector& v2)
{
    if(v1.magval() > v2.magval())
        return v1;
    else
        return v2;
}
```

- Returning an object invokes the copy constructor, whereas returning a reference doesn’t. Thus version 2 does less work and is more efficient.
- The reference should be to an object that exists when the calling function is executing.
- Both v1 and v2 are declared as being const references, so the return type has to be const to match.

2.3.2. Returning a reference to non-const object

Two common examples of returning a non-const object are overloading the **assignment operator** and overloading the **<< operator** for use with **cout**. The first is done for reasons of efficiency, and the second for reasons of necessity.

```
String& String::operator=(const String& st)
{
    if(this == &st)
        return *this;

    delete [] str;
    len = st.len;
    str = new char[len + 1];
    std::strcpy(str,st.str);

    return *this;
}
```

The return value of operator=() is used for chained assignment.

```
String s1("Good stuff");
String s2,s3;
s3 = s2 = s1;
```

Returning a reference allows the function to avoid calling the String copy constructor to create a new String object. In this case, the return type is not const because the operator=() method return a reference to s2, which it does modify.

The return value of operator<<() is used for chained output

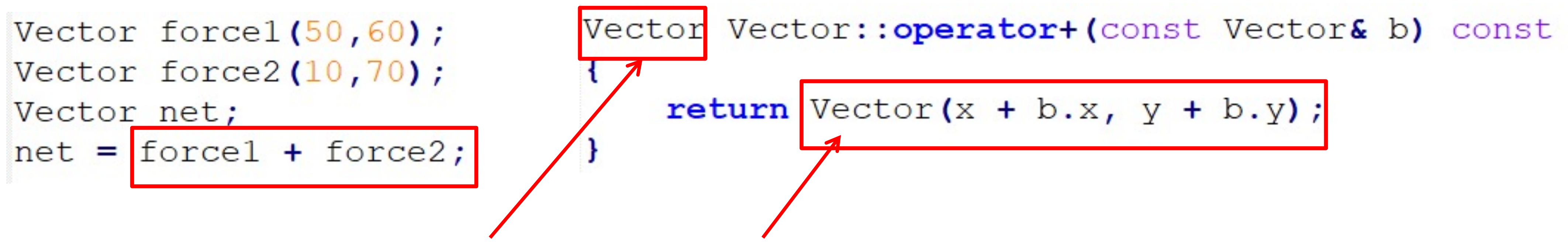
```
ostream& operator<<(ostream& os, const String& st)
{
    os << st.str;
    return os;
}
```

The return type has to be ostream & and not just ostream.The ostream class does not have a public copy constructor.

```
String s1("Good stuff");
cout << s1 << " is coming! ";
```

2.3.3. Returning an object

If the object being returned is local to the called function, then it should not be returned by reference because the local object has its destructor called when the function terminates. Thus, when control return to the calling function, there is no object left to which the reference can refer.



```
Vector force1(50,60);
Vector force2(10,70);
Vector net;
net = force1 + force2;
```

```
Vector Vector::operator+(const Vector& b) const
{
    return Vector(x + b.x, y + b.y);
}
```

The sum is a new, temporary object computed in Vector::operator+(), and the function shouldn't return a reference to a temporary object either. Instead, it should return an actual Vector object, not a reference.

There is the added expense of calling the copy constructor to create the returned object, but that is unavoidable.

2.3.4. Returning an const object

The definition of Vector::operator+() allows these two usage as follows:

```
net = force1 + force2;

force1 + force2 = net;
```

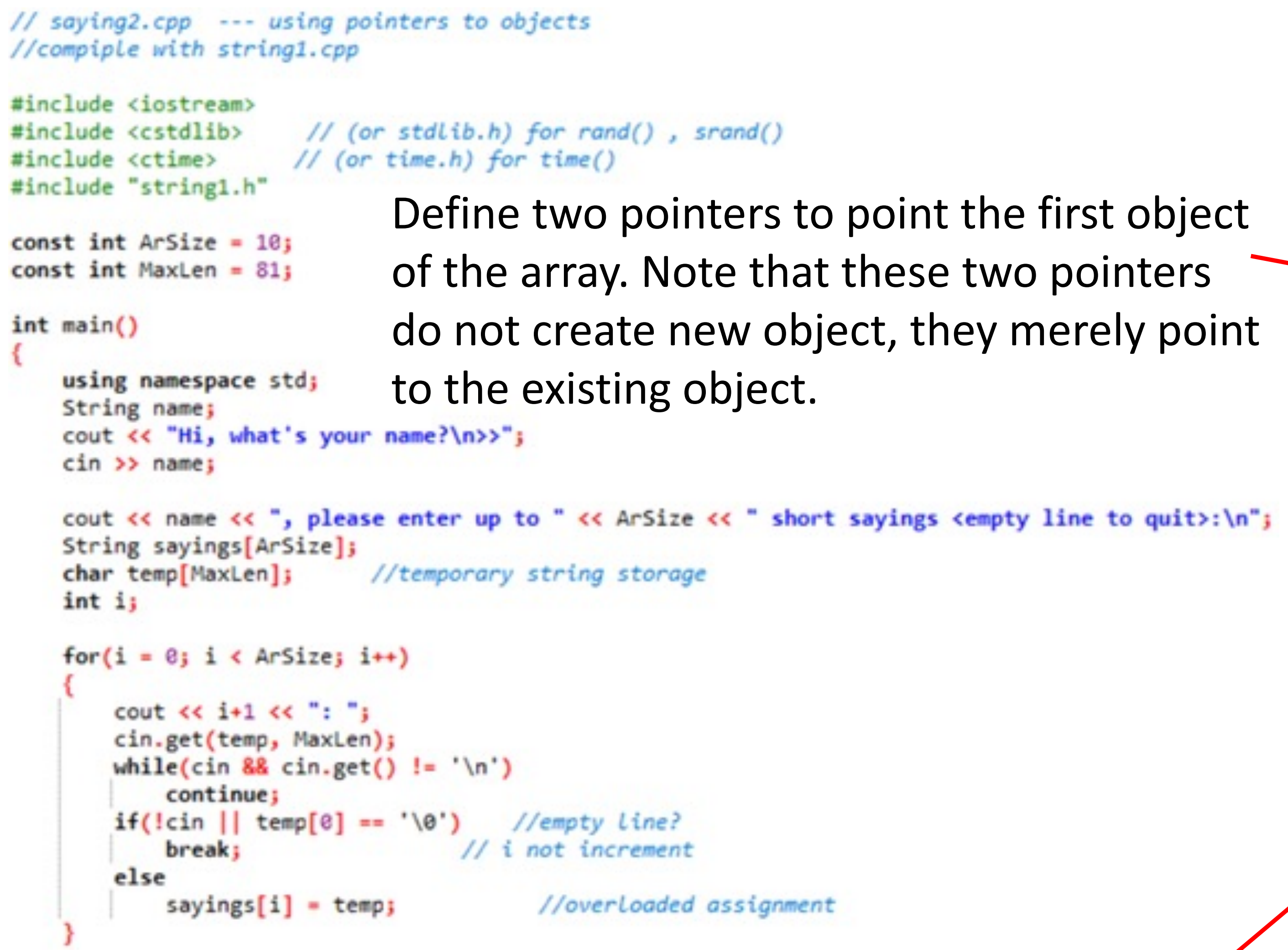
The expression force1 + force2 stands for the temporary object which the copy constructor constructs. In the statement 1, the temporary object is assigned to net, but in statement 2, the sum of force1 and force2 is assigned to an temporary object. This causes misuse.

```
const Vector Vector::operator+(const Vector& b) const
{
    return Vector(x + b.x, y + b.y);
}
```

Declare the return type as a const object. Then statement 1 is still allowed but the statement 2 becomes invalid.

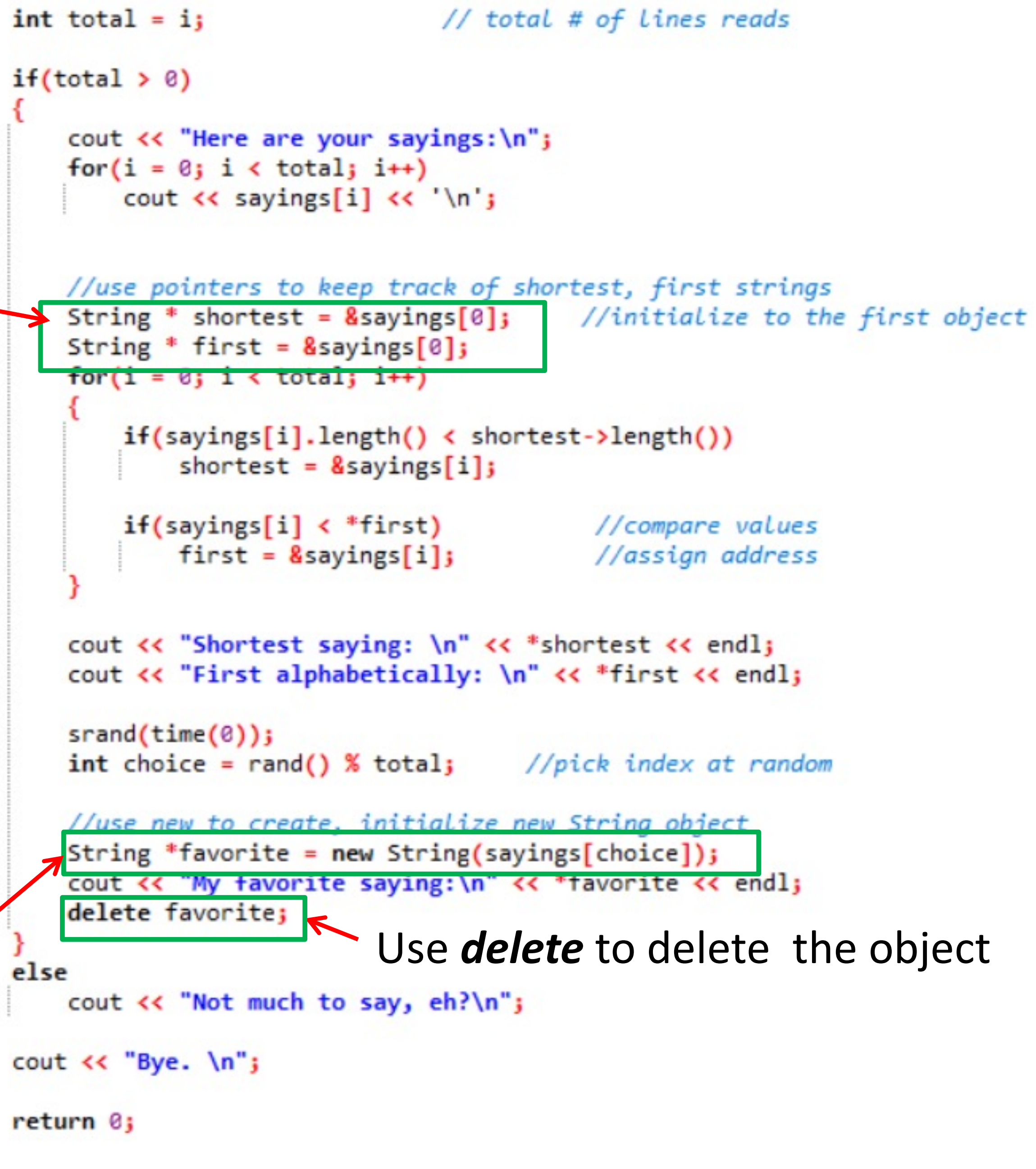
2.4 Using Pointers to Objects

C++ programs often use pointers to objects.



```
// saying2.cpp  --- using pointers to objects
//compiple with string1.cpp

#include <iostream>
#include <cstdlib>      // (or stdlib.h) for rand() , srand()
#include <ctime>       // (or time.h) for time()
#include "string1.h"

const int ArSize = 10;
const int MaxLen = 81;

int main()
{
    using namespace std;
    String name;
    cout << "Hi, what's your name?\n>>";
    cin >> name;

    cout << name << ", please enter up to " << ArSize << " short sayings <empty line to quit>:\n";
    String sayings[ArSize];
    char temp[MaxLen];        //temporary string storage
    int i;

    for(i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get(temp, MaxLen);
        while(cin && cin.get() != '\n')
            continue;
        if(!cin || temp[0] == '\0')    //empty line?
            break;               // i not increment
        else
            sayings[i] = temp;          //overloaded assignment
    }
```

Define two pointers to point the first object of the array. Note that these two pointers do not create new object, they merely point to the existing object.



```
int total = i;                  // total # of lines reads

if(total > 0)
{
    cout << "Here are your sayings:\n";
    for(i = 0; i < total; i++)
        cout << sayings[i] << '\n';

    //use pointers to keep track of shortest, first strings
    String * shortest = &sayings[0];    //initialize to the first object
    String * first = &sayings[0];
    for(i = 0; i < total; i++)
    {
        if(sayings[i].length() < shortest->length())
            shortest = &sayings[i];

        if(sayings[i] < *first)         //compare values
            first = &sayings[i];        //assign address
    }

    cout << "Shortest saying: \n" << *shortest << endl;
    cout << "First alphabetically: \n" << *first << endl;

    srand(time(0));
    int choice = rand() % total;     //pick index at random

    //use new to create, initialize new String object
    String *favorite = new String(sayings[choice]);
    cout << "My favorite saying:\n" << *favorite << endl;
    delete favorite;
}
else
    cout << "Not much to say, eh?\n";

cout << "Bye. \n";

return 0;
}
```

Use ***delete*** to delete the object

The pointer ***favorite*** provides the only access to the nameless object created by ***new*** and initializes the new ***String*** object by using the object saying[choice]. That invokes the copy constructor.

Several points about using pointers to objects:

1. Declare a pointer to an object by the usual notation:

```
String * glamour;
```

2. Initialize a pointer to point to an existing object:

```
String * first = &sayings[0];
```

3. Initialize a pointer by using *new*. Invoke the appropriate class constructor to initialize the newly created object:

```
// invokes default constructor
String *gleep = new String;

// invokes the String(const char *) constructor
String *glop = new String("Hell world");

// invokes the String(const String &) constructor
String *favorite = new String(sayings[choice]);
```

4. Use the *-> operator* to access a class method via a pointer:

```
if (sayings[i].length() < shortest->length())
```

5. Apply the *dereferencing operator(*)* to a pointer to an object to obtain an object:

```
if (sayings[i] < *first)        // compare object values
    first = &sayings[i];
```

3 Exercises

1. Rewrite the Stock class introduced in lecture10,program example **stock02.h**, to use dynamically allocated memory directly instead of using string class objects to hold the stock names. Also replace the show() member function with an overloaded operator<<() definition. Write a program to test the new definition.

```
// stock02.h -- augmented version
#ifndef STOCK20_H_
#define STOCK20_H_
#include <string>
class Stock
{
    private:
    std::string company;
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }

    public:
    Stock(); // default constructor
    Stock(const std::string & co, long n = 0, double pr = 0.0);
    ~Stock(); // do-nothing destructor

    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show()const;
    const Stock & topval(const Stock & s) const;
};
#endif
```

2. The declaration of Stack as follows:

```
// stack.h -- class declaration for the stack ADT
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};        // constant specific to class
    Item  * pitems;         // holds stack items
    int size;               // number of elements in stack
    int top;                // index for top stack item
public:
    Stack(int n = MAX);     // creates stack with n elements
    Stack(const Stack & st);
    ~Stack();
    bool isempty() const;
    bool isfull() const;
    // push() returns false if stack already is full, true otherwise
    bool push(const Item & item); // add item to stack
    // pop() returns false if stack already is empty, true otherwise
    bool pop(Item & item);  // pop top into item
    Stack & operator=(const Stack & st);
};
```

Implement all the methods and write a program to demonstrates all the methods, including copy constructor and assignment operator.