Operator Overloading	Operator overloading using Friend Function	The copy constructor and overloading the assignment operator	Shallow and deep copy

C++ Overloading (Operator and Function)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

Overloaded declarations have the same name with different arguments and obviously different definition (implementation). When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function overloading in C++:

The definition of the overloaded functions (same name) must differ from each other by the **types and/or the number** of arguments in the argument list. You cannot overload function declarations that differ only by **return type**. Following is the example where same function **print()** is being used to print different data types:

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

class printData
{
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};
```

```
int main(void)
{
  printData pd;
  pd.print(5);     // Call print to print integer
  pd.print(500.263);          // print float
  pd.print("Hello C++");     // print character
  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Operators overloading in C++:

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as:

1. ordinary non-member functions or 2. as class member functions.

In case we define above function as **non-member function of a class** then we would have to pass two arguments for each operand as follows: **Box operator+(const Box&, const Box&);**

Note : What is on the left-hand side of the "+" is the first argument and what is on the right-hand side of the "+" is the second argument in the operator+ function.

Here, we can view the code: array3=array1 + arrray2; as being rewritten by the compiler as: array3=operator+(array1, array2);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument, the object which will call this operator can be accessed using **this** operator as explained below:

```
class Box
{
  public:
    double getVolume(void)          {       return length * breadth * height;     }
    void setLength( double len )    {       length = len;         }
    void setBreadth( double bre )   {       breadth = bre;      }
    void setHeight( double hei )    {       height = hei;         }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b)    {
          Box box;              box.length = this->length + b.length;
          box.breadth = this->breadth +     b.breadth;        box.height = this->height + b.height;
          return box;
    }
  private:
    double length;       // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Main function for the program
int main( )
{    // Declare Box1, Box2, Box3  of type Box
     Box Box1;               Box Box2;          Box Box3;
     double volume = 0.0;    // To Store the volume of a box here
      // box 1 specification
```

```
   Box1.setLength(6.0);          Box1.setBreadth(7.0);        Box1.setHeight(5.0);

    // box 2 specification
    Box2.setLength(12.0);        Box2.setBreadth(13.0);      Box2.setHeight(10.0);

    // Volume of box 1
     volume = Box1.getVolume();          cout << "Volume of Box1 : " << volume <<endl;
    // Volume of box 2
     volume = Box2.getVolume();          cout << "Volume of Box2 : " << volume <<endl;

     Box3 = Box1 + Box2;      // Add two Box objects
     // volume of box 3
     volume = Box3.getVolume();          cout << "Volume of Box3 : " << volume <<endl;
    return 0;
}
```

When the above code is compiled and executed, it produces following result:

Volume of Box1 : 210 Volume of Box2 : 1560 Volume of Box3 : 5400

Overloadable/Non-overloadableOperators:

Following is the list of operators which can be overloaded:

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

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

:: (Scope Resolution)	.* (Pointer-to-member)	. (Member Access or Dot)	?: (Ternary or Conditional)	sizeof (Object size)	typeid (Object type)

Why those operators cannot be overloaded ?

- Above operators are close to the core of the language, allowing overloading of these operators can cause many problems/confusions without any benefits or it is syntactically not possible.

- For instance the sizeof operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you cannot overload it with your own runtime code.

- Scope resolution and member access operators work on names rather than values. C++ has no syntax for writing code that works on names rather than values so syntactically these operators cannot be overridden.

- Also, all operators that can be overloaded must have at least one argument that is a user-defined type. That is, operator overloading can apply only to your own classes.
    - means you can't overload operator+ for int or double.
    - means you can't overload the scope resolution operator since it has no arguments.
    - means you can't invent new operators.

Example :

```
// vectors: overloading operators example
#include <iostream>
using namespace std;
class CVector {
        public:
            int x,y;
            CVector () {};
            CVector (int,int);
            CVector operator + (CVector);
};
CVector::CVector (int a, int b) {  x = a;     y = b; }

CVector CVector::operator+ (CVector param) {
     CVector temp;
     temp.x = x + param.x;      temp.y = y + param.y;
     return (temp);
}
```

```
int main () {
     CVector a (3,1);
     CVector b (1,2);
     CVector c;
     c = a + b;
     cout << c.x << "," << c.y;
     return 0;
}
4,3
```

When two parameters are sent in non-member function a (left side operand) indicates the 1st parameter and b (right side operand indicates 2nd parameter.

Operator overloading function can be called either implicitly using the operator, or explicitly using the function name:
c = a + b; c = a.operator+ (b);
Both expressions are equivalent.

For non-member function it is
Either c= a+ b or c = operator+(a,b)

Note : CVector operator+ (CVector& param1, CVector& param2, CVector& param3) { }
If we pass **three** parameters in the binary operators it will give compile error.

- A class includes a **default** constructor and a **copy constructor** even if they are not declared,
- it also includes a default definition for the **assignment operator** (=) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:
  CVector d (2,3); CVector e; e = d; // copy assignment operator

- The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.
- The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use operator + to subtract two classes or operator== to fill with zeros a class, although it is perfectly possible to do so.

Unary Operator Overloading Example

The unary operators operate on a single operand and following are the examples of Uninary operators:

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--. Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
using namespace std;
class Distance
{
  private:
     int feet;             // 0 to infinite
     int inches;           // 0 to 12
  public:
     Distance(){      feet = 0;          inches = 0;        }       // required constructors
     Distance(int f, int i){   feet = f;    inches = i;  }

     // Method to display distance
     void displayDistance()    {     cout << "F: " << feet << " I:" << inches <<endl;     }


     // Overloaded minus (-) operator
     Distance operator- ()     {
              feet = -feet;       inches = -inches;       return Distance(feet, inches);
     }
};
int main()
{
       Distance D1(11, 10),    D2(-5, 11);                -D1;                    // apply negation
       D1.displayDistance();     // display D1
       -D2;                           // apply negation      D2.displayDistance();    // display D2
        return 0;
}
```

Result: F: -11 I:-10   F: 5 I:-11

Binary operators overloading Example

You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator. Following example explain how addition (+) operator can be overloaded. Similar way you can overload subtraction (-) and division (/) operators. In earlier Box example we have seen the overloading of binary operator +.

Relational operators overloading

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) You can overload any of these operators, which can be used to compare the objects of a class. Following example explain how a < operator can be overloaded and similar way you can overload other relational operators.

```
class Distance
{
    private:
        int feet;          int inches;
    public:
        Distance()              {    feet = 0;          inches = 0;       }
        Distance(int f, int i)  {    feet = f;          inches = i;       }
        // Method to display distance
        void displayDistance()       {
                cout << "F: " << feet << " I:" << inches <<endl;
        }
        // Overloaded minus (-) operator
        Distance operator- ()    {
            feet = -feet;        inches = -inches;          return Distance(feet, inches);
        }
        // Overloaded < operator
        bool operator <(const Distance& d)       {
            if(feet < d.feet)       {       return true;          }
            if(feet == d.feet && inches < d.inches)      {    return true;        }
            return false;
        }
};
int main()
{
     Distance D1(11, 10), D2(5, 11);
     if( D1 < D2 )     {   cout << "D1 is less than D2 " << endl;     }
    else               {   cout << "D2 is less than D1 " << endl;     }
    return 0;
}
```

It produces result:
D2 is less than D1

Overloading Increment ++ and Decrement --

Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way you can overload operator (--).

```
class Time
{
      private:
            int hours;       int minutes;         // 0 to 59
      public:
            Time()     {       hours = 0;       minutes = 0;     }     // required constructors
            Time(int h, int m)    {       hours = h;       minutes = m;       }
            // Method to display time
            void displayTime()     {    cout << "H: " << hours << " M:" << minutes <<endl;      }
            // Overloaded prefix ++ operator
           Time operator++ ()       {
                  ++minutes;         // increment this object
                  if(minutes >= 60)      {      ++hours;       minutes -= 60;       }
                  return Time(hours, minutes);       }
            // Overloaded postfix ++ operator
            Time operator++( int )     {
                  // Save the orignal value
                  Time T(hours, minutes);
                  ++minutes;                  // Increment this object
                  if(minutes >= 60)       {      ++hours;       minutes -= 60;       }
                  // Return old original value
                  return T;
             }
};
```

The C++ standard says that **if it has a dummy int parameter**, it's a postfix operator. The compiler uses the int argument to distinguish.

```
int main()
{
  Time T1(11, 59), T2(10,40);

  ++T1;              T1.displayTime();      // display T1
  ++T1;              T1.displayTime();      // display T1

  T2++;              T2.displayTime();      // display T2
  T2++;              T2.displayTime();      // display T2
  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

Operator overloading using Friend Function

s s1,s2;
s1 = 10+s2;

Here '+' is a binary operator which takes two operands , one is 10 which is an integer type and another operand s2 which is an object of class s

- The statement **s1=10+s2;** -> **10.operator+(s2)**. Here left hand side operand is 10 –integer data type with that we cannot invoke the operator() – member function.
- From this it is clear that if left hand side operand is not an object, then we cannot implement the **operator function as a member function**. So this statement should be implemented as a friend function.

Example : We have a rectangle class and we want to increase the length and width by user supplied value and for that we want to overload + operator

```
#include <iostream>
using namespace std;
class rect
{
    public:
       int width, length;
       rect ()    { width = length =0;  }
       void disp()        {cout << width <<" " << length;}
       void getdata()  {cin>> width >> length;  }
       friend  rect  operator+( int, rect);
};
rect operator+(int a, rect  rect1)
{
    rect  rect2;
    rect2.width = a + rect1.width;
    rect2.length = a + rect1.length;
    return rect2;
}
```

```
int main()  {
   rect  rct;     rct.getdata();
   rect  rct1 = 10+ rct;
   rct1.disp();
   return 0;
}
```

- First argument of operator+ is not an object so we have implemented it as a friend function.
- We need to pass two arguments for a binary operator. Because this pointer is not implicitly passed in the friend function.
- If width=7 and length=8, after creating this object it is passed to the operator function and value 10 is passed for an integer argument. So the output of the program is 17 18.

Overloading the operators >> and << using friend function

cin >>a; - the operator >> takes two operands, one is an object of **istream** and another is an **integer** data type.

If we want this to implement for an user defined type, then the code will be

Number s; cin>> s;

s is an object of class **Number** and cin is an object of **istream**. In this statement left hand side operand

is not an object of user defined class Number. So the operator >> cannot be overloaded using the member function of class Number. It should be implemented as friend function.

Example

```
#include <iostream>
using namespace std;
class s
{
    int i,j;
    public:
       s(){ i=j=0;}
       friend istream&  operator>>(istream&,s&);
       friend ostream& operator<<(ostream&,s&);
};
istream& operator>>(istream& in, s& s1)
{
    in >> s1.i >> s1.j;
}
ostream& operator<<(ostream& out, s &s1)
{   out << s1.i <<s1.j;   }
```

```
int main()
{
    s s1, s2;
    cin>> s1 >>s2;
    cout <<s1 << s2;
    getch();
    return 0;
}
```

In this program the operators >>, << are overloaded as a friend function to perform input output operations with objects. So friend function declarations have been mentioned inside the class.

This friend function takes istream and class s as an argument and it implies that it needs to work with objects of istream and class s. Inside the function definition the statement

Inside the main function the statement **cin>>s1>>s2;** calls the **oeperator>>** function for performing the input operation of a class s. The same way output operation also implemented using the operator function with ostream object and object of s.
ostream&operator<<(ostream& out, s &s1).

In this second example **getdata()** and **disp()** functions are replaced with operator functions **<<** and **>>** for reading input and output data members.

Instead of implementing **getdata()** and **disp()** functions we have implemented the input and output operations using operator functions. Operator functions are also functions. So what is the benefit of making it as a operator overloaded functions?

The copy constructor and the assignment operator

- In every class, the compiler automatically supplies both a copy constructor and an assignment operator if we don't explicitly provide them.
- Both of these member functions perform copy operations by performing a member wise copy from one object to another.
- Declaring a copy constructor does not suppress the compiler-generated copy assignment operator, nor vice versa. If you implement either one, we recommend that you also implement the other one so that the meaning of the code is clear.
- If a new object has to be created before the copying can occur, the copy constructor is used.
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

Copy constructor

- A copy constructor is a special constructor for a class/struct that is used to make a copy of an existing instance. According to the C++standard, the copy constructor for MyClass must have one of the following signatures:

```
MyClass( const MyClass& other );
MyClass( MyClass& other );
MyClass( volatile const MyClass& other );
MyClass( volatile MyClass& other );
```

Note that none of the following constructors, despite the fact that they *could* do the same thing as a copy constructor, are copy constructors:

```
MyClass( MyClass* other );
MyClass( const MyClass* other );
MyClass( MyClass other );
```

Why copy constructor argument should be const in C++?

- One good reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified.
- But there is more to it. For example, predict the output of following C++ program. Assume that copy elision (Copy elision or Copy omission is a compiler optimization technique that avoids unnecessary copying of objects. Now a days, almost every compiler uses it) is not done by compiler.

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

class Test
{
  /* Class data members */
  public:
      Test(Test &t) { /* Copy data members from t*/}
      Test()      { /* Initialize data members */ }
};
Test fun()   //  Return Test object
{
   cout << "fun() Called\n";   Test t;    return t;   }
```

const is not used

```
int main()
{
   Test t1;
   Test t2 = fun();
   return 0;
}
```

Here copy constructor is called to create t2 with return temporary Test object.

Compiler Error in line "Test t2 = fun();"

Return t by value, so a temporary Test object is created and return to main program.

The program looks fine at first look, but it has compiler error. If we add const in copy constructor, the program works fine, i.e., we change copy constructor to following.

```
Test(const Test &t) { cout << "Copy Constructor Called\n"; }
```

Or if we change the line "Test t2 = fun();" to following two lines, then also the program works fine.

```
Test t2;
t2 = fun();
```

- The function fun() returns by value. So a temporary object is created by fun() and returned to main which is then copied to t2 using copy constructor in the original program (The temporary object is passed as an argument to copy constructor). The reason for compiler error is, **compiler created temporary objects cannot be bound to non-const references** and the original program tries to do that. It doesn't make sense to modify compiler created temporary objects as they can die any moment.

  Normally, a temporary object lasts only until the end of the full expression in which it appears. Here the temporary returned by f() lives until the closing curly brace. However, C++ deliberately specifies that binding a temporary object to a reference ***to const*** on the stack lengthens the lifetime of the temporary to the lifetime of the **reference** itself, and thus avoids what would otherwise be a common dangling-reference error.

When do I need to write a copy constructor?

- First, you should understand that if you do not declare a copy constructor, the compiler gives you one implicitly. The implicit copy constructor does a member-wise copy of the source object. For example, given the class:

```
class MyClass {
    int x;
    char c;
    std::string s;
};
```

The compiler-provided copy constructor is exactly equivalent to:

```
MyClass::MyClass( const MyClass& other ) :   x( other.x ),
                                             c( other.c ), s( other.s )   {}
```

- In many cases, this is sufficient.
- However, there are certain circumstances where the member-wise copy version is not good enough.
- The most common reason the default copy constructor is not sufficient is because the object contains **raw pointers** and you need to take a "deep" copy of the pointer. That is, you don't want to copy the pointer itself; rather you want to copy what the pointer points to.
- Why do you need to take "deep" copies?

This is typically because the instance owns the pointer; that is, the instance is responsible for calling delete on the pointer at some point (probably the destructor). If two objects end up calling delete on the same non-NULL pointer, heap corruption results.

Assignment Operator

The **assignment operator** is used to copy the values from one object to another ***already existing object***. The key words here are "already existing". Consider the following example:

```
Cents cMark(5); // calls Cents constructor
Cents cNancy; // calls Cents default constructor

cNancy = cMark; // calls Cents assignment operator
```

The assignment operator must be overloaded as a member function.

The purpose of the copy constructor and the assignment operator are almost equivalent — both copy one object to another. However, the assignment operator copies to existing objects, and the copy constructor copies to newly created objects.

There are three general cases where the copy constructor is called instead of the assignment operator:

1. When instantiating one object and initializing it with values from another object (as in the example above).
2. When passing an object by value.
3. When an object is returned from a function by value.

- In each of these cases, a new variable needs to be created before the values can be copied — hence the use of the copy constructor.
- Because the copy constructor and assignment operator essentially do the same job (they are just called in different cases), the code needed to implement them is almost identical.

An overloaded assignment operator and copy constructor example

Now that you understand the difference between the copy constructor and assignment operator, let's see how they are implemented. For simple classes such as our Cents class, it is very straightforward. Here is a simplified version of our Cents class:

```
class Cents
{
    private:
        int m_nCents;
    public:
        Cents(int nCents=0)  {  m_nCents = nCents;   }
        // Copy constructor
        Cents(const Cents &cSource)   {    m_nCents = cSource.m_nCents;    }
};
```

Two things are worth explicitly mentioning.

1. As copy constructor is a member of Cents, and our parameter is a Cents, we can directly access the internal private data of our parameter.
2. The parameter MUST be passed by reference, and not by value. Can you figure out why?

A copy constructor is called when a parameter is **passed by value**. If we pass our cSource parameter by value, it would need to call the copy constructor to do so. But calling the copy constructor again would mean the parameter is passed by value again, requiring another call to the copy constructor. This would result in an infinite recursion (well, until the stack memory ran out and the the program crashed). Fortunately, modern C++ compilers will produce an error if you try to do this:

```
C:\\Test.cpp(431) : error C2652: 'Cents' : illegal copy constructor: first
                    parameter must not be a 'Cents'
```

The first parameter in this case must be a reference to a Cents! **Now let's overload the assignment operator.**

```
class Cents
{
    private:
        int m_nCents;
    public:
        Cents(int nCents=0)   {  m_nCents = nCents;    }
        // Copy constructor
        Cents(const Cents &cSource)    {  m_nCents = cSource.m_nCents;    }
        Cents& operator= (const Cents &cSource);
};
Cents& Cents::operator= (const Cents &cSource)
{
   m_nCents = cSource.m_nCents;        // do the copy
   return *this;                       // return the existing object
}
```

- The line that does the copying is exactly identical to the one in the copy constructor. This is typical. In order to reduce duplicate code, the portion of the code that does the actual copying could be moved to a private member function that the copy constructor and overloaded assignment operator both call.
- We're returning *this so we can chain multiple assigments together:

```
cMark = cNancy = cFred = cJoe; // assign cJoe to everyone
```

Finally, it is possible in C++ to do a self-assignment: `cMark = cMark;` // valid assignment

In these cases, the assignment operator doesn't need to do anything (and if the class uses dynamic memory, it can be dangerous if it does). It is a good idea to do a check for self-assignment at the top of an overloaded assignment operator. Here is an example of how to do that:

```
Cents& Cents::operator= (const Cents &cSource)
```

```
{
   // check for self-assignment by comparing the address of implicit object and the parameter
   if (this == &cSource)        return *this;
   // do the copy
   m_nCents = cSource.m_nCents;
   return *this;     // return the existing object
}
```

Note that there is no need to check for self-assignment in a copy-constructor. This is because the copy constructor is only called when new objects are being constructed, and there is no way to assign a newly created object to itself in a way that calls to copy constructor.

Default member wise copying : Because C++ does not know much about your class, the default copy constructor and default assignment operators it provides are very simple. They use a copying method known as a member wise copy (also known as a **shallow** copy).

Shallow copying

Because C++ does not know much about your class, the default copy constructor and default assignment operators it provides use a copying method known as a shallow copy (also known as a memberwise copy). A **shallow copy** means that C++ copies each member of the class individually using the assignment operator. When classes are simple (eg. do not contain any **dynamically allocated memory**), this works very well.

However, when designing classes that handle **dynamically allocated memory**, member wise (shallow) copying can get us in a **lot of trouble**! This is because the standard pointer assignment operator just copies the address of the pointer — it does not allocate any memory or copy the contents being pointed to! Let's take a look at an example of this:

```
class MyString
{
    private:
       char *m_pchString;
       int m_nLength;

    public:
       MyString(char *pchString="")
       {
           // Find the length of the string
           //  Plus one character for a terminator
           m_nLength = strlen(pchString) + 1;

           // Allocate a buffer equal to this length
```

Simple string class allocates memory to hold a string that we pass in. We have not defined a copy constructor or overloaded assignment operator. So, C++ will provide a default copy constructor and default assignment operator that do a shallow copy. Now, consider the following snippet of code:

```
MyString cHello("Hello, world!");
{
   // use default copy constructor
   MyString cCopy = cHello;
} // cCopy goes out of scope here
std::cout << cHello.GetString() << std::endl;
// this will crash
```

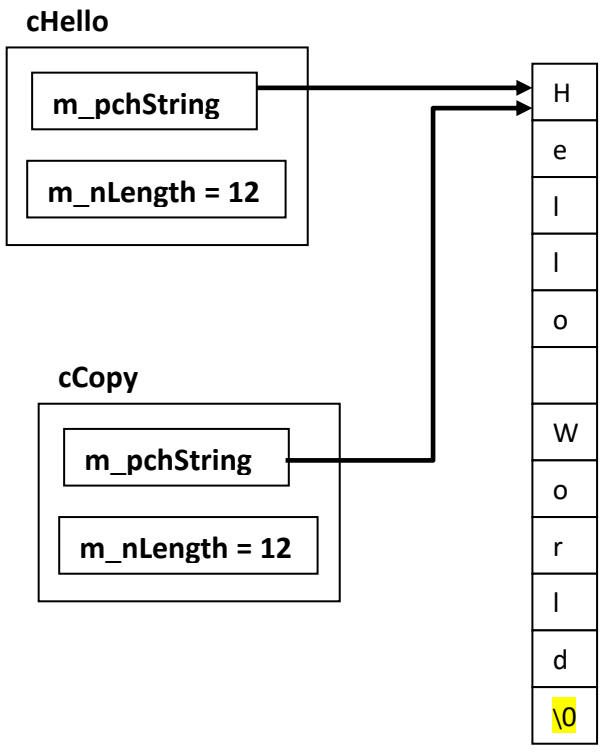
```
        m_pchString= new char[m_nLength];
        strncpy(m_pchString, pchString, m_nLength);      // Copy the parameter into our internal buffer
        m_pchString[m_nLength-1] = '\0';                 // Make sure the string is terminated

    }
    ~MyString()     // destructor
    {
        // We need to deallocate our buffer
        delete[] m_pchString;
         // Set m_pchString to null just in case
        m_pchString = 0;
    }
    char* GetString() { return m_pchString; }
    int GetLength() { return m_nLength; }
};
```

Now let's analyze the reason of the problem in the above code :

- **MyString cHello("Hello, world!");** - This line calls the MyString constructor, which allocates some memory, sets **cHello.m_pchString** to point to it, and then copies the string "Hello, world!" into it.

- **MyString cCopy = cHello;** // use default copy constructor

It's actually the source of our problem! When this line is evaluated, C++ will use the default copy constructor (because we haven't provided our own), which does a shallow pointer copy on **cHello.m_pchString**. Because a shallow pointer copy just copies the address of the pointer, the address of cHello.m_pchString is copied into **cCopy.m_pchString**. As a result, cCopy.m_pchString and cHello.m_pchString are now both pointing to the same piece of memory!



- } // cCopy goes out of scope here

In this line when cCopy goes out of scope, the MyString destructor is called on **cCopy**. The destructor deletes m_pchString are pointing to! Consequently, by deleting cCopy, we've also

(inadvertently) affected cHello. Note that the destructor will set cCopy.m_pchString to 0, but cHello.m_pchString will be left pointing to the deleted (invalid) memory!

- std::cout << cHello.GetString() << std::endl; // this will crash

As we deleted the string that cHello was pointing to, and now we are trying to print the value of memory that is no longer allocated.

The root of this problem is the shallow copy done by the copy constructor — **doing a shallow copy on pointer values in a copy constructor or overloaded assignment operator is almost always asking for trouble.**

Problems can also occur with shallow copying when we:

- initialize an object with the value of another object : name s1; name s2(s1);
- pass an object by value to a function or when we return by value : name function_proto (name)
- assign one object to another: s1 = s2;

Deep copying

To resolve the pass by value and the initialization issues, we must write a copy constructor whenever dynamic member is allocated on an object-by-object basis. They have the form:

class_name(const class_name &class_object);

- Notice the name of the "function" is the same name as the class, and has no return type
- The argument's data type is that of the class, passed as a constant reference

A **deep copy** duplicates the object or variable being pointed to so that the destination (the object being assigned to) receives it's **own local copy**. This way, the destination can do whatever it wants to it's local copy and the object that was copied from will not be affected. Doing deep copies requires that we write our own copy constructors and overloaded assignment operators. Let's go ahead and show how this is done for our MyString class:

```
// Copy constructor
MyString::MyString(const MyString& cSource)
{
   // because m_nLength is not a pointer, we can shallow copy
   m_nLength = cSource.m_nLength;
    // m_pchString is a pointer, so we need to deep copy it
   if (cSource.m_pchString)
   {
```

check to make sure cSource even has a string. If it does, then we allocate enough memory to hold a copy of that string. Finally, we have to manually copy the string using strncpy() .

```
        // allocate memory for our copy
        m_pchString = new char[m_nLength];

        // Copy the string into our newly allocated memory
        strncpy(m_pchString, cSource.m_pchString, m_nLength);
    }
    else
        m_pchString = 0;
}
```

Now let's do the overloaded assignment operator. The overloaded assignment operator is a tad bit trickier:

```
// Assignment operator
MyString& MyString::operator=(const MyString& cSource)
{
    // check for self-assignment
    if (this == &cSource)        return *this;

    delete[] m_pchString;    // first we need to deallocate any value that this string is holding!

    m_nLength = cSource.m_nLength;    // because m_nLength is not a pointer, we can shallow copy it

    // now we need to deep copy m_pchString
    if (cSource.m_pchString)
    {
        m_pchString = new char[m_nLength];        // allocate memory for our copy

        // Copy the parameter the newly allocated memory
        strncpy(m_pchString, cSource.m_pchString, m_nLength);
    }
    else
        m_pchString = 0;

    return *this;
}
```

Note that our assignment operator is very similar to our copy constructor, but there are three major differences:

- We added a self-assignment check
- We return *this so we can chain the assignment operator.
- We need to explicitly deallocate any value that the string is already holding.

When the overloaded assignment operator is called, the item being assigned to may already contain a previous value, which we need to make sure we clean up before we assign memory for new values. For non-dynamically allocated variables (which are a fixed size), we don't have to bother because the new value just overwrite the old one. However, for dynamically allocated variables, we need to explicitly deallocate any old memory before we allocate any new memory. If we don't, the code will not crash, but we will have a memory leak that will eat away our free memory every time we do an assignment!

Checking for self-assignment

- In our overloaded assignment operators, the first thing we do is check for self assignment. There are two reasons for this.

1. One is simple efficiency: if we don't need to make a copy, why make one?
2. The second reason is because not checking for self-assignment when doing a deep copy will cause problems if the class uses dynamically allocated memory.

Let's take a look at an example of this. Consider the following overloaded assignment operator that does not do a self-assignment check:
If we do not check for self-assignment what happens when we do the following?
cHello = cHello;

This statement will call our overloaded assignment operator. The *this* pointer will point to the address of cHello (because it's the left operand), and cSource will be a reference to cHello (because it's the right operand). Consequently, m_pchString is the same as cSource.m_pchString.

Now look at the first line of code that would be executed: delete[] m_pchString;.

This line is meant to deallocate any previously allocated memory in cHello so we can copy the new string from the source without a memory leak. However, in this case, when we delete m_pchString, we also delete cSource.m_pchString! We've now destroyed our source string, and have lost the information we wanted to copy in the first place. The rest of the code will allocate a new string, then copy the uninitialized garbage in that string to itself. As a final result, you will end up with a new string of the correct length that contain garbage characters.

The self-assignment check prevents this from happening.

Another Example of Operator overloading

You can make operators **virtual** just like other member functions. Implementing **virtual** operators often becomes confusing, however, because you may be operating on two objects, both with unknown types. This is usually the case with mathematical components (for which you often overload operators). For example, consider a system that deals with matrices, vectors and scalar values, all three of which are derived from class **Math**:

```
//: C15:OperatorPolymorphism.cpp
// Polymorphism with overloaded operators
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual Math& multiply(Scalar*) = 0;
  virtual Math& multiply(Vector*) = 0;
  virtual ~Math() {}
};

class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};

class Scalar : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
```

```
class base {
public:
 virtual int operator+(int) const = 0;
};

class deriv: public base {
public:
 int operator+(int) const {return 23;}
};

int main() {
 deriv d;      base& b = d;
 std::cout << b + 15 << std::endl;
 return 0;
}
```

If we compile and runs it gives 23 as expected.

For simplicity, only the **operator*** has been overloaded. The goal is to be able to multiply any two **Math** objects and produce the desired result – and note that multiplying a matrix by a vector is a very different operation than multiplying a vector by a matrix.

```
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};

class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      m1 * m2;
    }
} ///:~
```

The problem is that, in **main( )**, the expression **m1 * m2** contains two upcast **Math** references, and thus two objects of unknown type. A virtual function is only capable of making a single dispatch – that is, determining the type of one unknown object. To determine both types a technique called *multiple dispatching* is used in this example, whereby what appears to be a single virtual function call results in a second virtual call. By the time this second call is made, you've determined both types of object, and can perform the proper activity. It's not transparent at first, but if you stare at the example for awhile it should begin to make sense.