CS-1201 Object Oriented Programming

Operator Overloading

Arbish Akram

Department of Computer Science
Government College University

Operator overloading

- Operator overloading allows custom implementations of operators for user-defined types (classes).
- Commonly overloaded operators include arithmetic, comparison, and assignment operators.
- C++ allows most existing operators to be overloaded.
- Two types of operators:
  - Unary operators: Operate on one operand (e.g., '-', '++').
  - Binary operators: Operate on two operands (e.g., '+', '-').

Example

```
class Counter
 {
 private:
     unsigned int count;
 public:
     Counter() : count(0)
         { }
     unsigned int get_count()
         { return count; }
     void operator ++ () {
         ++count;
     }
 };
 int main()
 {
     Counter c1, c2; //define and initialize
     court << "\nc1=" << c1.get_count();
     court << "\nc2=" << c2.get_count(); //display
     ++c1;  //increment c1
     ++c2;  //increment c2
     court << "\nc1=" << c1.get_count(); //display again
     court << "\nc2=" << c2.get_count() << endl;
     return 0;
 }
```

- In this program, we create two objects of the class ‘Counter‘: ‘c1‘ and ‘c2‘.
- The counts in these objects are displayed initially; they are set to ‘0‘.
- Using the overloaded ‘++‘ operator, we increment ‘c1‘ and ‘c2‘.
- Finally, we display the resulting values.

```
c1 = 0
c2 = 0
c1 = 1
c2 = 1
```

The operator Keyword

- The operator keyword is used to overload a C++ operator for user-defined types.
- Overloading the '++' operator with the following declaration:

```
void operator ++ ()
```

- The return type is followed by the keyword operator, the operator itself ('++'), and an argument list.
- This tells the compiler to call this member function when the '++' operator is used with a Counter object.

Overloading and Data Types

- The compiler distinguishes between overloaded functions based on the number and types of their arguments.
- Similarly, it distinguishes between overloaded operators based on the data type of their operands.

```
++intvar;  // Uses built-in routine for int \\
++c1;      // Uses user-defined operator++() for Counter
```

- If the operand is a basic type (e.g., ‘int‘), the built-in routine is used.
- If the operand is of a user-defined type (e.g., `Counter`), the user-written `operator++()` is called.

Operator Return Values

- The `operator++()` function, as defined in last example has a 'void' return type.
- This can cause issues in assignment statements like:

  ```
  c1 = ++c2;
  ```

- The compiler expects a `Counter` type to be returned, but the current definition returns `void`.
- As defined, '++' can't be used in assignment statements; it must be a standalone operation.

Improving `operator++()`

- To use '++' in assignment expressions, we need to modify `operator++()` to return a value.
- Overloading operators requires careful consideration of return types, especially when the operator is expected to work in assignment contexts.

Example

```
class Counter
 {
 private:
     unsigned int count;
 public:
     Counter() : count(0)
         { }
     unsigned int get_count()
         { return count; }
     Counter operator ++ () //increment count
         {
         ++count; //increment count
         Counter temp; //make a temporary Counter
         temp.count = count; //give it same value as this obj
         return temp; //return the copy
         }
 };

```

Example

```
int main()
   {
       Counter c1, c2;  //c1=0, c2=0
       court << "\nc1=" << c1.get_count();
       court << "\nc2=" << c2.get_count();//display
       ++c1; //c1=1
       c2 = ++c1;  //c1=2, c2=2
       court << "\nc1=" << c1.get_count(); //display again
       court << "\nc2=" << c2.get_count() << endl;
       return 0;
           }
```

Overloading Binary Operators: Example 1

```
class Box {
    double length, breadth, height;
    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+(Box b) {
          Box box;
          box.length = length + b.length;
          box.breadth = breadth + b.breadth;
          box.height = height + b.height;
          return box;
       }
};
```

Overloading Binary Operators: Example 1

```cpp
int main() {
   Box Box1, Box2, Box3;
   double volume = 0.0;  // Store the volume of a box here

   Box1.setLength(1.0);
   Box1.setBreadth(4.0);
   Box1.setHeight(5.0);
   Box2.setLength(2.0);
   Box2.setBreadth(3.0);
   Box2.setHeight(5.0);

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

   // Add two object as follows:
   Box3 = Box1 + Box2;

   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
   return 0;
}
```

Overloading Binary Operators: Example 1

```
Volume of Box1 : 20
Volume of Box2 : 30
Volume of Box3 : 210
```

Overloading Binary Operators: Example 2

```
class Distance
{
private:
    int feet;
    float inches;
public:
//constructor (no args)
Distance() : feet(0), inches(0.0)
    { }
//constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
    { }
void getdist() //get length from user
{
    court << "\nEnter feet: "; cin >> feet;
    court << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "\'-" << inches << '\"'; }
Distance operator + ( Distance ) const; //add 2 distances
};
```

Overloading Binary Operators: Example 2

```cpp
Distance Distance::operator + (Distance d2) const //return sum
{
    int f = feet + d2.feet;   //add the feet
    float i = inches + d2.inches; //add the inches
    if(i >= 12.0)  //if total exceeds 12.0,
    {
        i -= 12.0; //then decrease inches by 12.0 and
        f++; //increase feet by 1
    }
return Distance(f,i); //return a temporary Distance
}
int main()
{
    Distance dist1, dist3, dist4; //define distances
    dist1.getdist();   //get dist1 from user
    Distance dist2(11, 6.25); //define, initialize dist2
    dist3 = dist1 + dist2; //single ‘+’ operator
    dist4 = dist1 + dist2 + dist3; //multiple ‘+’ operators
    court << "dist1 = ";
    court << "dist2 = ";
    court << "dist3 = ";
    court << "dist4 = ";
    return 0;
}
```

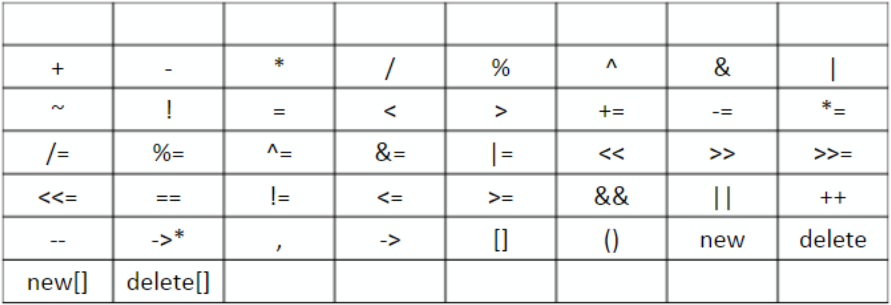
Concatenating Strings

```cpp
#include <iostream>
using namespace std;
#include <string.h> //for strcpy(), strcat()
#include <stdlib.h> //for exit()
class String   //user-defined string type
{
private:
    enum { SZ=80 };   //size of String objects
    char str[SZ];    //holds a string
public:
    String() //constructor, no args
        { strcpy(str, ""); }
    String( char s[] )   //constructor, one arg
        { strcpy(str, s); }
    void display() const //display the String
        { cout << str; }
    String operator + (String ss) const //add Strings
    {
        String temp;  //make a temporary String
        if( strlen(str) + strlen(ss.str) < SZ )
        {
            strcpy(temp.str, str);  //copy this string to temp
            strcat(temp.str, ss.str); //add the argument string
```

Concatenating Strings

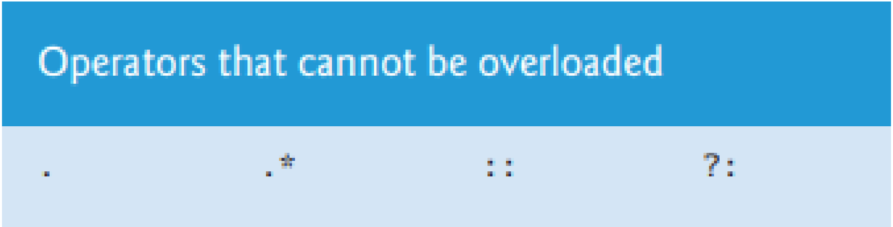
```cpp
 }
        else
        { court << "\nString overflow"; exit(1); }
    return temp; //return temp String
    }
};
int main()
{
    String s1 = "\nMerry Christmas! ";   //uses constructor 2
    String s2 = "Happy new year!";   //uses constructor 2
    String s3;   //uses constructor 1
    s1.display();
    s2.display();
    s3.display();   //display strings
    s3 = s1 + s2;  //add s2 to s1, assign to s3
    s3.display();
    cout << endl;
    return 0;
}
```

Operators That Can Be Overloaded



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

Operators That Cannot Be Overloaded



Operators that cannot be overloaded			
.	.*	::	?: