Initialization and Assignment

Chapter 7

Objectives

- Define and use overloaded operators in your code.
- Explain the semantics of assignment.
- Distinguish between initialization and assignment.
- Overload the assignment operator.
- Implement type conversions by overloading cast operators and by constructors.
- Gain experience through code walk-throughs and lab exercises

  - The example programs are in the **chapter directory.**
  - Labs located in Labs/Lab7

Operator Overloading

- **Part of the strategy of making an abstract data type is to make it as much like a built-in data type as possible.**
- **Function:**
  - d = Sub(Add(a, b), c);
- **Operator:**
  - d = a + b - c;
- Almost all C++ operators may be overloaded. An original name may *not* be created for an operator function.
- Standard associativity rules remain valid:
  - operator = is right associative, operator + is left associative.
- An operator function must take *at least one class argument*.
  - This prevents a programmer from changing the behavior of built-in data types.

Operator Functions

- **The function name is of the form operator *op* where *op* is a standard C operator such as +, *, ->, [], etc.**
- **It is called by using "infix" operator notation, e.g.**
  - a **op** b (for a binary operator)
- **Operands are arguments of operator function or invoking object, in case of a member function.**
- **Value of operator expression is value returned by function.**

Concatenation Demo

- **String/Step0 contains a string class with a function *Concat*, which we will convert to an operator.**
- **Create a new project then add the files to your project.**
- **Build and run the program.**
- **The only change required is to replace *Concat* with *operator+* in class specification and implementation, and to change the usage to infix +.**
- **Make these changes and then build and run the program again.**

Semantics of *return*

- **The operator *+* returns an instance of *String* class. The *return* statement initializes an area of memory provided by the function's caller to hold the object returned by the function.**
  - This memory area may be a temporary object created by the compiler.
  - The **return** statement initializes the returned object using the copy constructor.
- **What happens if no copy constructor is provided?**
  - The **return** statement will copy the pointer from instance in called function to instance in calling function.
  - The instance in the called function will be destroyed, deallocating memory pointed to by the returned pointer. This leaves the pointer in the calling program pointing to deallocated memory.

Returning a Temporary Object

- **Look closely at the code for *operator+***

```
String String::operator+(const String& s) {
          String temp(buf);   // temporary
          ...
          return temp;
}
```

- **This code is valid because we are returning a value, which gets created by the copy constructor as part of the return mechanism.**
- **IMPORTANT - It would be invalid to return a reference in this case, because the calling program would have a reference to a memory object that no longer exists!**

Returning a Reference

- **In some cases it may be valid to return a reference.**
  - The data to be returned must still be valid after the return.
  - An example is a case where the invoking object itself gets updated as the return value.
- **This situation will be illustrated with the overloaded assignment operator.**
  - The left hand side is both the invoking object and the new value after the assignment.

Initialization vs. Assignment

- **Consider standard C initialization of a variable:**
  - int x = 5;
  - Variable x is created and initialized with value 5.
- **C++ provides alternate notation**
  - int x(5);
- **Alternative is first to define variable, without an initial value, and then do an assignment:**
  - int x;
  - x = 5;

Initialization vs. Assignment (continued)

- **With objects, initialization involves invoking a constructor:**

```
String s = "hello";       // (const char *) conversion constructor
String t;                 // default constructor
t = s;                    // assignment
```

Semantics of Assignment

- **A built-in assignment operator is available in every class.**
- **The built-in assignment operator does a "shallow" memberwise copy.**
- **If your class has pointer member objects, you should always implement your own overloaded assignment operator so that the data pointed to gets copied.**

Assignment

- **Override the assignment operator when your class has member data with dynamically allocated memory.**
- **The default assignment operator copies only the pointer, which may *appear* to work, but may introduce a bug!**
- **Examine behavior of default assignment operator in folder AssignmentBug.**

Assignment (continued)

- **The *m_str* pointer for both *a* and *b* will point to string *“Alpha".***
- **There is a problem:**
  - Data allocated for string **b** is never deallocated, yielding a memory leak.
  - If one of **a** or **b** goes out of scope, the **m_str** pointer in the other will be invalid.

Overloading =

```
String& String::operator=(const String& s){
   if (this == &s)       // special case s = s
      return *this;
   length = s.length;
   delete [] m_str;
   m_str = new char[length + 1];
   strcpy(m_str, s.m_str);
   return *this;
}
```

- Tip: *Use of reference return type avoids having to do a copy on return.*

String Assignment Demo

- String/Step1 **contains a String class which implements an assignment operator.**
- **Create a new project then add the files to your project.**
- **Build and run the program.**

Review of *this* Pointer

- **Each class member function contains a pointer of its class type named *this*.**
- **The *this* pointer contains the address of the class object through which the member function has been invoked.**
- **Hence **this* will refer to the invoking object itself.**

Type Conversions

- **When type conversion is appropriate for your class, implement it by constructors and overloaded cast operators.**
- **Use a constructor to convert from a standard type to a class type:**
  - String(const char *str); //convert from const char* to String
- **Use an overloaded cast operator to convert from a class type to a standard type (or to another class type without having to modify the definition of the other class):**
  - operator const char* () const;

Conversion by Construction

- **Another type can be converted to an object by a constructor.**
- **A character pointer is converted into a C++ String object.**
  - String(const char *str = "");
  - Note use of default argument, so that we do not need a separate default constructor.

Overloading Cast Operator

- **An object can be converted to another type by an overloaded cast operator:**

```
class String {
public:
   operator const char* () const;
   ...
};
String::operator const char* () const{
   return m_str;
}
```

The **const** grants read-only access to the internal data buffer associated with the **String** object

Demo

- **Folder Complex contains a program that demonstrates the use of overloaded MDAS operators for a complex type.**
  - In this example just multiplication is demonstrated
- **Build and run the program.**

Summary

- **Most C operators can be overloaded by using an *operator op* definition.**
- **Initialization creates a new object with a defined value, and assignment gives a new value to an existing object.**
- **The built-in assignment operator does a shallow member-wise copy.**
- **The = operator can be overloaded by a member function returning **this*.**
- **The built-in assignment operator should be overridden when there is dynamically allocated member data.**
- **Type conversions can be accomplished by constructors and by overloading cast operators.**