Object Oriented Programming and C++

Xiaoqiang Wang
Florida State University

Further aspects of C++

- Default arguments in functions
- Object construction
- Class data members: pointer, reference or object?
- Friends
- Function overloading
- Operator overloading
- Templates and Container Types

Default arguments in functions

In function prototypes you can specify default values for arguments.
The function declared as

```
int myFunction (int x);
```

requires an int argument when it is called, for example

```
myFunction (40)
```

However, if a default value is declared as below

```
int myFunction (int x = 50);
```

the function can be called without an argument.

```
myFunction ()
```

In this case the default value is used, and this is the same as

```
myFunction (50)
```

Default arguments

- Any or all of a function's arguments can have default values with the following restriction.
  - If one of the arguments does not have a default argument, no previous argument can have a default value.

int
BlockVolume (int width, int height, int length = 10); // Legal

int
BlockVolume (int width, int height = 20, int length = 10); // Legal

int
BlockVolume (int width = 10, int height = 20, int length = 10); // Legal

int
BlockVolume (int width, int height = 20, int length); // Illegal

int
BlockVolume (int width = 10, int height, int length = 10); // Illegal

Default arguments for class member functions

Example: The second constructor for the **String** class.

If we declare a default value of zero for the C string:

String (const char* s = 0);

this can act as the default constructor.

We can now get rid of the original default constructor.

Object construction

- There are two ways that data members can be set in constructors:
  - Assignment
  - Initialisation

- Consider the following class **Word**, which stores a **String** and the number of characters in the string.

```
class Word
{
        public:
        Word (const String& name = 0, int length = 0);
        private:
        String name_;
        int length_;
};
```

Assignment

- There are two ways the constructor can set the values of **name_** and **length_**

  (1) Through assignment in the body of the constructor

```
Word::Word(const String& name, int length)
{
    name_ = name; // assignment
    length_ = length; // assignment
}
```

Initialisation lists

(2) Through initialisation in an *initialisation list*

```
Word::Word (const String& name, int length)
: name_(name), // initialisation
length_(length) // initialisation
{
    // Nothing to do here
}
```

- Each data member to be initialised is listed after a : with an argument (which can be an expression) in parentheses.
- Each data member is separated by a comma.

Another point about initialisation

- If a class contains a data member of a type whose constructor requires parameters, that data member must be initialised in an initialisation list.

```
class X
{
        public:
                X(int n, int k); // Constructor
        private:
                A a;
                int j;
} ;

X::X(int n, int k)
        : a(n), // Calls A's constructor
        j(k)
{
        // Nothing to do
}
```

Initialising const and reference data members

- These can only be initialised through an initialisation list
- Example

```
class X
{
        public:
                X (int i);
        private:
        int i_;
        const int ci_;
        int& ri_;
};
```

Initialising const and reference data members

```
X::X (int i)
{
    i_ = i; // OK
    ci_ = i; // Error: can't assign to a const
    ri_ = i; // Error: ri_ is not initialised
}
```

Here the data members **ci_** and **ri_** must be initialised, not assigned.

```
X::X (int i)
    : i_(i), // OK
    ci_(i), // OK
    ri_(i) // will compile, but it's bad!
{
    // Nothing to do here
}
```

Reference argument

- Don't use an object passed by value to initialise a reference data member as this can result in a dangling reference when the formal argument goes out of scope.
- An argument used for this purpose must be passed by reference.
- Here is a slightly modified version of the above code

```
X::X (int& i) // Reference argument
        : i_ (i),
        ci_ (i),
        ri_ (i) // ri_ initialised with i
{
        // Nothing to do here
}
```

Be careful!

- **List members in an initialisation list in the order in which they are declared.**

```
class Array
{
    public:
        Array (int lowBound, int highBound);
    private:
        int* data; // Pointer to data
        unsigned int size; // Number of elements
        int lowBound_,; // Lower bound
        int highBound_; // Upper bound
};
```

Initialisation order

- Using this initialisation list, data is not initialised correctly.

```
Array::Array (int lowBound, int highBound)
    : size (highBound-lowBound+1),
    lowBound_(lowBound),
    highBound_(highBound),
    data(new int[size]),
{
    // Nothing to do
}
```

- Because data members are initialised in the order of their *declaration in the class*, not their *order in the initialisation list*.
- In this example, data is initialised before size.
- Solution: change the declaration order.

Programming recommendations

- Use initialisation lists for consistency and efficiency.
- Initialise data members in the order in which they are declared in the class declaration.
- For consistency, use initialisation lists for all data members, including built in types.
  - Because of the use of initialisation lists, the bodies of constructors may be empty, or almost empty.

Class data members: pointer, reference or object?

- If class **X** contains a member function **y** of type **Y**, should **y** be declared as

  Y y; // an object

  or **Y* y; // a pointer**

  or **Y& y; // a reference**

Guidelines

- It is usually simplest and preferable to use an object (**Y y;**) because then memory management is automatic.

  Use a pointer if the object pointed to can change during the lifetime of the containing object.

  Example: class **String**

  The representation uses a pointer

  char* data;

  This needs to be changed during assignment: deleted and re-created to accommodate strings of different lengths.

Guidelines

- Only use a reference if

  1. **y** refers to an existing object when **X** is created
  2. an **X** object makes no sense if the **Y** member does not previously exist
  3. **y** refers to the same **Y** object for the lifetime of the **X** object

  Example: A **Call** class can exist to associate two existing **Telephone** objects

```
class Call
{
    public:
    ...
    private:
        Telephone& caller;
        Telephone& callee;
};
```

Here, both the caller and callee must exist before a Call object is created.

An alternative would be to use ordinary objects, but that would require creating local copies of both caller and callee every time a call is made.

That would be inefficient.

Friends

- What is a friend?
- A friend of a class is either
  - a non-member function
  - another whole class
  - a member function of another class that has access to the private and protected members of the class.
- We will only discuss friend non-member functions

Friend declaration

- The friend declaration begins with the keyword **friend**.
- This can only appear in the declaration of the class.
- Friendship is thus granted by the class, not unilaterally grabbed by the friend.
- A friend function does not have access to the **this** pointer.
- This has implications for operator overloading, as discussed below.

Example

```
class X
{
    public:
        friend void
        FriendSetData (X& x, int b);

        void
        MemberSetData (int b);
    private:
        int data;
};

void
FriendSetData (X& x, int b) {x.data = b;}

void
X::MemberSetData (int b) {data = b;}
```

Example

```
int main (void)
{
    X x;

    FriendSetData (x, 10);
    x.MemberSetData (10);
}
```

- Notes:

1. The friend function has an extra argument (X& x) because it cannot use the implicit this pointer like the member function.
2. The friend function is not called with the x. notation, because it is not a member function of x (or any class).

Friend functions

- ***What should friend functions be used for?***

  To improve the public interface of a class.

- ***Where should friend functions be declared in a class****?*

  In the **public** section.

  The compiler ignores the **public** , **protected**, or **private** access specifiers for friend functions.

Function signatures

Examples

int f1 (int, double);
int f2 (int, const char*, const char*);

The names of the parameters are irrelevant when determining the signatures, and do not have to be declared in the prototypes.

Non-**const** and **const** parameters are considered distinct as far as the signature is concerned.

int f3 (int, double);
int f4 (const int, double);

Function Overloading

Two or more functions in the *same scope* can be given the same name provided each signature is unique.

Such functions are said to be **overloaded**.

For example

```
int Max (int, int);
int Max (const int*, int);
int Max (const List&);
```

Each function requires a separate implementation, but from the user's point of view there is only one operation: each returns the largest of its arguments.

Function Overloading

```
int m = Max (j, k);
int n = Max (intArray, 1024);
```

Without the ability to overload a function name each implementation would have to be given its own unique name:

```
int IntMax (int, int);
int IntArryMax (const int*, int);
int ListMax (const List&);
```

This is lexically more complex.

Function Overloading

It is the compiler's responsibility to call the correct function.

How does do this?

The simplest way is when it finds an exact match between the supplied arguments and the parameters, as in the above example.

The parameters have to be sufficiently different for overloading to work.

For example

int f (int);
int f (int&); // Error: int and int& not // sufficiently different

Function Overloading

- A function cannot be overloaded purely on the basis of its return type

```
int f (int);
double f (int); // Error: can't differ only in return
    type
```

- It is common to overload the constructors of class.

Operator Overloading

- The designer of a class can provide a set of operators to work with objects of the class.
- This is done by overloading (a large subset of) the predefined C++ operators.
- The name of an operator function consists of the keyword **operator** followed by one of the C++ operators: **operator@** where @ is a C++ operator.
- Operator overloading is just *syntactic sugar*, which allows us to use notation like **a + b** instead of **a.operator+ (b)**, which is what we are really doing.

Operator Overloading

- An operator function can either be a member function of the class or a non-member function.
- If it is a non-member function and needs to use the private members of the class it must be a **friend**.
- It must take at least one argument that is the same as the class (except **new** and **delete**).

- The following C++ operators can be overloaded

+ - * / % ^ & |

~ ! , = < > <= >=

++ -- << >> == != && ||

+= -= /= %= ^= &= |= *=

<<= >>= [] () -> ->* new delete

Things you can't do

- You can't overload the following operators

  :: .* . ?:
- You can't overload the predefined meaning of the operators for the built in types.
- You can't add new operators to the built in types.
- You can't change the *arity* of the operators. The arity of an operator is the number of arguments it takes.

- A *binary* operator takes two arguments.
- A *unary* operator takes one argument.

- For example you can't define a unary **%** or a binary **!**

- You can't define new operator tokens, eg ****** for exponentiation.
- This would complicate compilation.

Binary Operators

- A binary operator can be defined as either a member function taking one argument or a non-member function taking two arguments.
- For example, for the binary operator **+**, the expression **a + b** can be interpreted as either

  a.operator+ (b) // For a member function

  or

  operator+ (a,b) // For a non-member function

Unary Operators

- A unary operator can be defined as either a member function taking no arguments or a non member function taking one argument.
- For example, for the prefix unary operator **+**, the expression **+a** can be interpreted as

  a.operator+ () // For a member function

  or

  operator+ (a) // For a non-member function

Examples

```
// Non-member functions (often friends)
X operator- (X); // prefix unary minus

X operator- (X,X); // binary minus

X operator- (); // error: no operand

X operator- (X, X, X); // error

X operator% (X); // error: defines a unary % operator
```

The String class

```
class String
{
    public:
        String (const char* s = 0); // Constructor
        String (const String& s); // Copy constructor
        String& operator= (const String& s); // Assignment
        String operator+ (const String& s); // Concatenation
        char& operator[] (int element); // Element
        friend ostream& operator<< (ostream& os, String& s); //
                                                          //Display
        int length (void) const; // length
        ~String (void); // Destructor

    private:
        char* data;
};
```

Concatenation

```
String String::operator+ (const String& s)
{
    String temp;
    delete[ ] temp.data;
    temp.data = new char [strlen (data) +
        strlen(s.data) + 1];
    strcpy (temp.data, data);
    strcat (temp.data, s.data);
    return (temp);
}
```

Concatenation

- The **String** class needs to support the following three forms of concatenation
  - **String** + **String**
  - **String** + "a C string"
  - "a C string" + **String**

- What about

  "a C string" + "a C string" ?

Main function

```
int main (void)
{
    String a (“Scientific");
    String b (" Programming");
    String c;
    String d;
    c = a + b;
    cout << c << endl;
    d = a + " Programming";   // implicitly convertion
    cout << d << endl;
    return (0);
}
```

Output:

```
Scientific Programming
Scientific Programming
```

Note

In the expression **d = a + " Programming“**

the C string **" Programming "** is *implicitly converted* to a **String**

because the constructor

String (const char* s = 0); // Constructor

takes a C string as an argument.

The constructor is implicitly called by the compiler.

This is also known as *user defined conversion*.

You therefore don't need a second concatenation operator

String operator+ (const char* s)

which takes a C string as an argument, although this would be more efficient.

Question

Now add the following code to the main function above

String e;
e = “I’m here" + b;
cout << d << endl;

This results in a compilation error message, because an overloaded operator which is a **String** member function must have a **String** as the first argument.

That is, **a + b** is really **a.operator+(b)**, and **“I’m here".(b)** makes no sense.

For this case we do require another function.

Solution

Use a non-member function with two **String** arguments:

```
String operator+ (const String& a, const String& b);
```

However, this must be a **friend** function, because it uses the private data member **data**.

```
class String
{
    ...
    friend
    String operator+ (const String& a, const String& b);
    ...
};
```

Non-member friend function

```
String
operator+ (const String& a, const String& b)
{
    String temp;
    delete[] temp.data;
    temp.data= new char
    [strlen (a.data) + strlen (b.data) + 1];
    strcpy (temp.data, a.data);
    strcat (temp.data, b.data);
    return (temp);
}
```

Note: (remember no friend keyword in the header)

Non-member friend function

This works with all three forms of concatenation expressions, including

e = “Scientific" + b;

where the compiler can now implicitly convert **" Scientific "** to a **String**.

The **String** class thus does not need a member function for concatenation.

Here the **friend** mechanism has been used to improve the user interface of the string class.

The operator [ ]

- Users of the **String** class need read and write access to the individual characters of the **data** class member.
- We would like to be able to do things like the following

```
String name ("kevin");
String buffer;
...
cout << name [0] << endl;

name [0] = 'K';
...
buffer [index] = name [index];
```

The operator [ ]

- The **[]** operator must be able to appear on the left and right sides of an assignment.
- To do this is must return a reference to a char.

```
char&  String::operator[ ] (int index)
  {
       return (data [index]);
  }
```

return as a reference

```
string a = " a b c d";
printf a[0];
a[0] = 'x';
// a == " x b c d "
```

Overloading <<

- The left operand of **<<** must be of class **ostream** , so the function that overloads **<<** must be a non-member function
- Since it requires access to the private data member of **String**, it must be a **friend**.

```
ostream& operator<< (ostream& os, String& s)
{
   return (os << s.data);
}
```

Member, non-member, or friend functions?

- If an operator requires a left operand of type other than the class, it must be a non-member.
- If it requires access to the non public members of the class, it must be a friend.
- The operators **=** (assignment), **[]** (subscript), and **()** (function call) must be member functions.
- When there is a choice, it is usually best to make it a member function.

Templates and Container Types

- To discuss class templates in C++

Container classes

- Container classes hold collections of objects of a particular type
  - Array
  - Linked list
  - Stack
  - Queue
  - Trees - eg binary trees

IntArray

- Consider an array class for storing **int**'s

```
class IntArray
{
    ...

    private:
    int* elements;
};
```

- This particular class can only store int's.

CharArray

- If we wanted an array for storing **char**'s, we would have to declare another class

```
class CharArray
{
    ...

    private:
    char* elements;
};
```

PersonArray

- Similarly if we wanted an array to store one of our own classes, **Person** for example, we would require

```
class PersonArray
{
        ...

        private:
        Person* elements;
};
```

- This can be a lot of work.

Templates

- Templates allow us to write a single class (such as a container class) that can hold elements of arbitrary type at run time.
- They provide *parametric polymorphism*.
- It is best to write and debug a class for holding a specific type before writing it as template class.
- Templates also allow us to write functions that work with arbitrary data types, for example a Quicksort funtion that can sort an array of any type of objects on which there is defined comparison operators **< <= > >=**

Example - Array

- *Before the* **Array** *declaration:* Place the line **template<class Type>**
  - Here **Type** is a *substitution parameter*.
  - Although **template** is a keyword, **Type** is not.
  - When you declare an array to hold a specific type, the compiler will substitute your type for **Type**.
- *Inside the **Array** declaration:* Everywhere the array declaration refers to the type of information stored in the array it is replaced by **Type**.

Example - Array

- *Outside the **Array** declaration:*
  - The definition of every member and non-member function of the class must be preceded by

    template<class Type>
  - The name of the class, **Array**, must be replaced by **Array<Type>,** exceptions next.
  - Return Type: change the corresponding type to **Type.**
  - When your define an array, you must specify the type of elements it is to hold between <> brackets:
    - **Array<int> intArr;**
    - **Array<float> floatArr;**
    - **Array<Complex> cArr;**
    - **Array<String> sArr;**

Example - Array

```
#include <iostream.h>

template<class Type>
^^^^^^^^^^^^^^^^^^^^
class Array
{
	public:
			Type& operator[](int index);
			^^^^
	private:
			enum {numElements = 100};
			Type theElements [numElements];
			^^^^
};


template<class Type>
^^^^^^^^^^^^^^^^^^^^
Type& Array<Type>::operator[] (int index)
^^^^ ^^^^^^
{ return (theElements[index]);}
```

```
int main()
{
    Array<int> intArr;
    ^^^^^
    for (int j = 0; j < 5; j++)
        intArr[j] = j * j;
    for (int j = 0; j < 5; j++)
        cout << intArr[j] << '\t';
    cout << endl;
    Array<float> floatArr;
    ^^^^^^^
    for (int j = 0; j < 5; j++)
        floatArr[j] = j + 2.25;
    for (int j = 0; j < 5; j++)
        cout << floatArr[j] << '\t';
    return (0);
}

// Output
0 1 4 9 16
2.25 3.25 4.25 5.25 6.25
```

Function Definition

- The class name **Array** must be replaced by **Array<Type>**

```
Array&
Array::operator= (const Array& rhs)
{
    …
}
```

Change to

```
template <class Type>

Array<Type>&
Array<Type>::operator= (const Array<Type>& rhs)
{
    …
}
```

Exceptions

- The names of the
  - constructors
  - copy constructor
  - destructor

  We only use **Array** for these, not **Array<Type>**

```
template <class Type>

Array<Type>::Array (const Array<Type>& rhs)
: numElements(rhs.numElements),
theElements(0)
{
    AllocateMemory();
    Copy(rhs);
}
```

More than one template parameter

- You can have more than one template parameter:

  template<class Type1, Type2>

Acknowledgement

- Most Slides from Kevin Suffern.