Practical Session #03 - C++ Classes

Tom Mahler
In This Recitation We’ll Cover

- Parameter Passing
- Classes
- Objects
- The Rule Of 3 (Classes That Hold Pointers)
Parameter Passing
Parameter Passing

- Correct parameter passing can help reduce the number of bugs and generate more efficient code.
- Passing parameters can be done in 3 distinct methods:
  - Passing by value
  - Passing by pointer
  - Passing by reference
Examples

- See code...
In this example the primitive integer value is passed to the sqrt function by value.

This means that the value of the number is being copied to the argument list of the called function.

Passing by value is usually done for primitive variables.

Passing by value also works with objects, it will cause the compiler to add a call to the copy constructor making it a slow option.

In general, unless a new copy of the object is needed, we will never pass an object by value.
Passing By Pointer - Example

- The function must work on the same string that was sent to it, so if we used passing by value the change would not impact the string that was sent from the main function.

- Notice that when passing by pointer we can also pass pointers to NULL (which is 0).

- Passing by pointer is not actually real.

- A pointer is just a numerical value, similar to int, that holds the address of a parameter, we are actually passing the address by value.
Passing By Reference - Example

- The syntax is the same as passing by value, except for the "&" that declares word as a reference.
- Passing by reference allows us to send a “link” to the original object without using pointers.
  - This means that any change in the object word will also apply on mySentence.
- This is usually a much safer method since pointers might suggest responsibility for the memory allocation and release and might confuse when attempting to search for bugs.
- References must be initialized at start, this means that they cannot have a NULL value.
Passing By Reference - Example

- Behind the scenes, the compiler actually looks at references as constant pointers (cannot have their addresses changed)
- The compiler passes the address by value, similarly to passing by pointer, but it does not allow the programmer access to the address nor does it allow the user to change it
- This method should be used in most instances of object parameter passing
Classes
C++ Class Syntax VS Java Class Syntax: Declarations & Implementation Are Separated

- The class definition only contains the declarations of the methods
- The actual implementation are listed separately, where each method name is prefixed by the class name
- The :: operator separates class name and method name
- Separation of declaration and implementation is usually done in cases where you want to export or share the class with another class, and is not required for inner, local private or helper classes
C++ Class Syntax VS Java Class Syntax: Semicolon At The End Of The Class

- There is a semicolon at the end of the class declaration

- Not placing it will lead to unclear compilation error
C++ Class Syntax VS Java Class Syntax: Public & Private Section

- In C++, there are public, protected & private sections, started by the keywords public, protected & private

- In Java, each individual item must be tagged with public or private
C++ Class Syntax VS Java Class Syntax: Const Methods

- const functions do not change the state of an object
- Good candidates for const are accessors functions (getters)
- Furthermore const functions cannot be used in a way that would allow you to use them to modify const data
- This means that when const functions return references or pointers to members of the class, they must also be const
- Also notice, that const methods can only access const methods
- Once declared a const variable, one can only use methods declared const on that variable
Examples

See code...
In C++, we use a member initialization list to initialize class members.

The initial value can be any expression.

The member initialization list is executed before the body of the function.

The order the initialization happens is according to the order the member vars are declared (not the order in the member initialization list).

It is hence a convention to keep the order of the list as the order of the declaration.
Member Initialization List

- It is possible to initialize data members inside the constructor body but not advised for the following two reasons:
  - **Implicit call to default constructor**
    - When a data member is itself a class object, not initializing it via the initialization list means implicitly calling its default constructor!
    - If you do initialize it in the body of the constructor you are actually initializing it twice
    - If your data member is a class with no default constructor, meaning you supplied some constructor that has parameters, you will not be able to pass compilation - think why
  - **Const members**
    - Const members of a class can only be initialized via member initialization list
The declarations and the implementation are defined separately in C++

We place the class declaration file in a header file (e.g. X.h) and the class implementation in file X.cpp

To avoid including a Header twice, we check whether a pre-compiler unique variable is defined

If not we define it and include the header
A convention is to use as a variable the name of the header class

For example:

//Header file of Point class
#ifndef _POINT_H_
define _POINT_H_
// now the header class declaration as before
//...//
#endif
Objects
In C++, object variables hold values, not object references
- You simply supply the construction parameters after the variable name

Example:

Point p(1, 2); /* construct p */

If you do not supply construction parameters, then the object is constructed with the default constructor

Time now; /* construct now with Time::Time() */
Objects

- This is very different from Java
  - In Java, this command would merely create an uninitialized reference
  - In C++, it constructs an actual object

- When one object is assigned to another, a copy of the actual values is made
  - In Java, copying an object variable merely establishes a second reference to the object
  - Copying a C++ object is just like calling clone in Java
  - Modifying the copy does not change the original
Objects

Point q = p; /*invokes copy constructor*/

- In most cases, the fact that objects behave like values is very convenient

- There are, however, a number of situations where this behavior is undesirable
Objects

- When modifying an object in a function, you must remember to use call by reference.

- Two object variables cannot jointly access one object
  - If you need this effect in C++, then you need to use references.

- An object variable can only hold values of a particular type
  - If you want a variable to hold objects from different subclasses, you need to use pointers.

- If you want a variable point to either null or to an actual object, then you need to use pointers in C++.
The "this" Pointer

- As in Java, this is a pointer to the active object
  - So for example, when L1 = L2 is executed, L1's member function copy assignment operator is called, so this is a pointer to L1

- We also make use of this for the returned value in the copy assignment operator
  - The type to be returned is List& so we dereference this, a.k.a return *this
Examples

See code...
The "this" Pointer

- We would like to establish a way to distinguish between parameters and class variables
  - This can be done by the "this" pointer
  - It used as a pointer to the class object instance by the member functions

- this pointer stores the address of the object instance, to enable pointer access of the object

- this pointers are not accessible for static member functions

- this pointers are not modifiable
Operator -> ASnd Operator.

- The pointer-to-member operators, .* and ->*, return the value of a specific class member for the object specified on the left side of the expression.

- The right side must specify a member of the class.

- The following example shows how to use these operators.
Examples

See code...
The Rule Of 3 (Classes That Hold Pointers)

- A thumb rule in C++ (prior to C++11) for classes that own a resource

- This rule requires such class to define and implement the following functions:
  - A destructor
  - A copy constructor
  - A copy assignment operator
The Rule Of 3 Example

- We learned that a class that own a resource must explicitly define three functions, according to the rule of 3.

- We will now see an example of such classes, a List class and a Link that together implement a linked list object.

- The List class includes a pointer to a dynamically allocated Link.
The Rule Of 3 Example

- The List class own the resource head_, and the responsibility for allocating memory, deleting it and making changes to it belongs to the class.

- In a similar way, the Link class own the resource for the next_ object in the list.

- We will demonstrate declarations of the class's destructor, copy constructor, and assignment operator.
Examples

▶ See code...
Destructor

- The main purpose of the destructor function is to free any dynamically allocated storage pointed to only by a data member of that object.

- Note that it is up to the programmer to ensure that no other pointers are pointing to that storage.
Destructor

- An object's destructor function is called when that object is about to "go away"

- i.e., when:
  - A class instance (a value parameter or a local variable) goes out of scope, or
  - The dynamically allocated storage pointed to by the pointer is freed by the programmer using the delete operator
Examples

- See code...
In this example, the scope of value parameter L is the whole function.

L goes out of scope at the end of the function.

So when function f ends, L's destructor function is called.

Note:

- If f had one or more return statements, L's destructor function would be called when a return was executed!
- We passed L by value only for the sake of the example as it is not recommended, it should have been passed by reference.
Destructor

- The scope of variable L1 is the body of the while loop
- L1's constructor function is called at the beginning of every iteration of the loop, and its destructor function is called at the end of every iteration of the loop
- Note that if the loop included a break or continue statement, the destructor would still be called!
Destructor

- Variable p is a pointer to a List

- When a List object is allocated using new, that object's constructor function is called

- When the storage is freed, the object's destructor function is called (and then the memory for the List itself is freed)
Question: Is a destructor function of a reference parameter called at the end of the function?

- A reference parameter's destructor function is not called at the end of the function because the corresponding actual parameter refers to the same object.
- The object does not "go away" when the function ends, so its dynamically allocated storage should not be freed.
Destructor

- Destructor functions are defined using a syntax similar to that used for the constructor function (the name of the class followed by a double colon followed by the name of the function) and should always be declared as virtual.

- If you don't write a destructor function for a class that includes pointers to dynamically allocated storage, your code will still work, but you will probably have some storage leaks.
Examples

See code...
Copy Constructor

- An object's copy constructor is called (automatically, not by the programmer) when it is created, and needs to be initialized to be a copy of an existing object.

- This happens when an object is:
  - Passed as a value parameter to a function
  - Returned (by value) as a function result
  - Declared with initialization from an existing object of the same class
Copy Constructor

L2 = f( L1 )

- Here, variable L1 is passed as a value parameter to function f
- The corresponding formal parameter is L
- When the call is executed, L's copy constructor is called to initialize L to be a copy of the actual parameter, L1
Copy Constructor

List tmp1 = L;

- Here variable tmp1 is declared to be a List, initialized to be the same as variable L
- When this line is executed, tmp1's copy constructor is called to initialize tmp1 to be a copy of L
- Similarly, when the following line is executed, tmp2's copy constructor is called to initialize tmp2 to be a copy of L

List tmp2(L);
Copy Constructor

```
return tmp1;
```

- Here, variable tmp1 is returned as the result of calling function f
- When this line is executed, the copy constructor is called to make a copy of tmp1 to be returned
- Later, that copy is used as the right-hand side of the assignment on:

```
L2 = f( L1 );
```
Copy Constructor

- A Copy Constructor must exist on classes that manages resources

- If you don't write a copy constructor, the compiler will provide one that just copies the value of each data member (this is sometimes called a shallow copy)

- If some data member is a pointer, this causes aliasing (both the original pointer and the copy point to the same location), and may lead to trouble
The Copy Constructor Declaration

- The copy constructor has one argument: its type is the class, and it is a \texttt{const} reference parameter

- The argument is the object that the copy constructor is supposed to copy

\begin{verbatim}
List(const List &L); // copy constructor
\end{verbatim}
The Copy Constructor Definition

- The definition of the copy constructor (the actual code for the function) should be put in a ".cpp" file, along with the code for the other class member functions.

- The copy constructor should copy the values of all non-pointer data members, and should copy the objects pointed to by all pointer data members (this is sometimes called a deep copy).
Examples

▶ See code...
Consider this example:

Link l1, l2;
l1 = l2;  // this assignment is OK

By default, class assignment is just a field-by-field assignment (i.e., a shallow copy is done)

This assignment is equivalent to:

l1.data_ = l2.data_;
l1.next_ = l2.next_;
Copy Assignment Operator

- If a class includes pointer fields, the default assignment operator causes aliasing, which lead to trouble!

- The default assignment can also cause storage leaks when the class has a pointer field

- To prevent these problems, you should always override the assignment operator as a class member function for a class with a pointer field

- You may either define the operator to do a deep copy or make it private
Copy Assignment Operator

- The declaration of the member function looks like this for the List class:

```cpp
List & operator=(const List &l);
```

- When the assignment `l1=l2` is executed, `l1`'s member function operator is called, and `l2` is passed as the argument to that function.

- It can be actually be written like this `l1.operator=(l2)`. 
Copy Assignment Operator

- Note that List's operator function returns a List, in order to permit chained assignment.

- For example: l1 = l2 = l3;
  - When this statement is executed, the expression l2 = l3 is evaluated first.
  - The result of evaluating that expression is used as the right-hand side of the assignment to l1.
  - The operator function returns its result by reference (that's what the ampersand means).
  - This is done for efficiency, to prevent the List copy constructor being called to make a copy of the returned value.
  - So this can be written as l1.operator=(l2.operator=(l3)).
Copy Assignment Operator

- Copy assignment operator differs from the copy constructor in three important ways:
  - The object being assigned to has already been initialized, therefore, if it has a pointer field, the storage pointed to must be freed to prevent a storage leak
  - It is possible for a programmer to assign from a variable into itself
    - For example: \( l1 = l1; \)
    - The copy assignment operator code must check for this case, and do nothing
  - The copy assignment operator code must return a value
Copy Assignment Operator

Here is the definition of copy assignment operator for the List class

It should always include the following 4 sections:
- Check assignment to self
- Clear existing data members
- Copy data member from other
- Return *this
Examples

- See code...
Exception Safety In Copy Assignment Operator

- Suppose the l.copy() expression yields an exception (either because there is insufficient memory for the allocation or because Link’s copy constructor throws one), the List will end up holding a pointer to a deleted Link.

- Such pointers are toxic!
  - You can’t safely delete them
  - You can’t even safely read them
  - About the only safe thing you can do with them is spend lots of debugging energy figuring out where they came from...
Exception Safety In Copy Assignment Operator

- Happily, making copy assignment operator exception-safe typically renders it self-assignment-safe, too.

- In many cases, a careful ordering of statements can yield exception-safe (and self-assignment-safe) code.

- Here, for example, we just have to be careful not to delete the List pointed by head_ until l.copy() completed successfully.
Examples

See code...
Exception Safety In Copy Assignment Operator

- Now, if l.copy() throws an exception, head_ remains unchanged.
- Even without the identity test, this code handles assignment to self, because we make a copy of the original List, delete the original List, then point to the copy we made.
- It may not be the most efficient way to handle self-assignment, but it does work.
Exception Safety In Copy Assignment Operator

- If you're concerned about efficiency, you could put the identity test back at the top of the function.
- Before doing that, however, ask yourself how often you expect self-assignments to occur, because the test isn't free.
- It makes the code (both source and object) a bit bigger, and it introduces a branch into the flow of control, both of which can decrease runtime speed.
- The effectiveness of instruction prefetching, caching, and pipelining can be reduced, for example.