Object Oriented Programming in C++

CS 247

University of Waterloo

cs247@student.cs.uwaterloo.ca

May 16, 2014

Overview

The Rule of Three

The "Rule of Three" is that if a class defines a copy constructor, assignment operator or a destructor it should define all three. The reasoning behind it is that if a deep copy is required then the destructor will need to deallocate memory and the other way around.

Copy Constructors

The copy constructor for a class is a constructor that takes a reference to an object of the same class as its only parameter. If you do not include a copy constructor a default one will be provided that performs a shallow copy.

```
MyClass(const MyClass&);
```

Copy Constructors

There are three places where copy constructors are called:

- When creating a new object and initializing it with values from another object of the same type.
- When passing an object by value.
- When an object is returned from a function by value.

Example : copyCtor.cpp

Assignment Operators

The assignment operator (operator=) is used to copy one object into another existing object. If you do not declare an assignment operator a basic one which performs a shallow copy will be provided.

The Copy and Swap Idiom

The copy and swap idiom is a way of avoiding code duplication and easily made errors. Assignment operators may be implemented by creating a new object of the same type with the copy constructor then swapping the old values of the object being assigned to with the values in the newly created object.
Example : assignment.cpp

Initialization Lists

Initialization lists can be used as part of a constructor to set the values of variables within an object.

Initialization Lists

It may seem that setting values in an initilization list is no different than doing so in the body of the constructor, *however*, using the initialization list can be necessary for a few reasons:

- Any object not initialized in the initialization list will have its default constructor called which may cause unecessary computation or unwanted side effects.
- References and const members cannot be set outside the initilization list.

Example : initialization.cpp

Initialization Lists

One issue to be noted is that the values set in the initilization list are set in the order that they are declared in the class definition, not in the order they appear in the list. This can cause problems when attempting to set values based on the value of other variables being initialized.
Example : initList.cpp

Const

The C++ keyword const is a promise not to change the value stored by variables.

Const Variables

Constant variables cannot be assigned to.

```
const int* x
```

x is a pointer to a constant integer. *x cannot be changed, but x can.

```
int* const y
```

y is a constant pointer to an integer. y cannot be changed, but *y can.

```
const int* const x
```

z is a constant pointer to a constant integer. Neither z nor *z can be changed.

Scope of Constants

Even if defined outside of a pair of scope braces the scope of a constant will be limited to the file where it is defined by default. To avoid this the keyword extern is needed. In the file providing the constant:

```
extern const int var = 42;
```

In the file using the constant:

```
extern const int var;
```

Const Functions

A function declared as:

```
type MyClass::f() const;
```

promises not to change any values in the MyClass object it is called on. It may be understood as the "this" pointer being a pointer to a constant object.
Only const functions may be called on const objects.

Const References

Often variables should be passed to and returned from functions as const references. Passing by const reference provides:

- Increased efficiency when passing objects since the object does not need to be copied since it is a reference.
- A promise that the original object will not be changed despite being a reference since it is constant.
- The ability to pass literals and temporary values because the reference is constant.

Example : constRef.cpp

Type Conversion

Type conversion in C++ can occur either implicitly or explicitly using type casting.

Implicit Conversion

Implicit conversion can happen with:

- Single argument constructors
- Assignment operators
- Type-cast operators

Type Casting

Type casting can be accomplished with (relative) saftey using static_cast and dynamic_cast.

```
dynamic_cast<new_type>(expression)
static_cast<new_type>(expression)
```

Dynamic_cast is used with pointers and references to classes and will ensure the result is valid. Static_cast can also perform conversions between pointers to related classes, but will *not* check for validity. It can also be used to perform conversions allowed implicity and their opposites.
Example : conversion.cpp

The End