Smart Pointers and Modularity

CS 247

University of Waterloo

cs247@student.cs.uwaterloo.ca

June 6, 2014

Overview

1. Smart Pointers
   - auto_ptr
   - Shared Pointers

2. Modularity

Smart Pointers

Smart pointers are objects that act as pointers but require less explicit management by the programmer as a result of providing useful features such as bounds checking and automatic memory management.

auto_ptr

auto_ptr is a class template used for holding unique pointers to objects, to use it you need to include the library containing it.

```
#include <memory>
```

auto_ptr enforces ownership over what it points to, meaning that no two auto_ptrs may point to the same object. This prevents errors by making sure that when an auto_ptr goes out of scope and its destructor is called no dangling pointers remain. When auto_ptrs are correctly used they will deal with most memory issues, however, auto_ptrs cannot replace every pointer since often you will not want the ownership property.

unique_ptr

In C++11 auto_ptr has been replaced with unique_ptr, however, the school servers have not been updated to use C++11 so it may not be used in this course.

Shared Pointers

Often we do want to be able to have multiple pointers to the same object. In C++11 shared_ptr can be used for this purpose. We do not have shared_ptr in this course but we can create our own class of smart pointer to manage memory in such cases.
How can we know when to delete an object when there could be multiple pointers refering to it?

Reference Counting

We can keep track of the number of pointers to an object haing a running count that we increment whenever a new pointer gets the address and decrement when a pointer ceases to point to the address. When the reference count reaches 0, it is safe to delete the object.

Modularity

Modularity is the idea of making multiple distinct modules (independent components) that form a program. Programming in modules can allow increased code reuse, make it easier to track down bugs and reduces code duplication.
When using Object Oriented Programming an easy way to divide your program into modules is to create separate files for every class you define.

Cohesion

Cohesion is how closely related the elements in a module are. Having high cohesion makes modules easier to understand because they have fewer operations and makes them more reusable because the different parts of the module make sense to be used together.

Coupling

Coupling is how much a module relies on other modules. High coupling makes code harder to reuse, because it is difficult to separate it from other modules and it has less of a clear purpose of its own. Tight coupling also means changes in one module will often require other modules to be changed.

Include Statements

When code has been separated into modules existing in numerous files it is necessary to include header files so that you can make use of the modules in your program. By including the header file you essentially place its contents into the file in which it is included allowing you to make use of functions defined elsewhere that are declared in the header.
Do not include C++ source code files (e.g. .cpp, .cc), doing so can have unintended side effects and loses the advantages of having code spread over multiple files.

Makefiles

Makefiles allow you to give a set of instructions to compile your program. They not only make compiling more convenient by allowing you to run one short command but also make it more efficient by only compiling files which rely on source which has changed since their last compilation.
For this course and probably often outside of it you can simply copy the "smart" makefile used here and edit the names of the object code files. The makefile will derive the rules necessary to build each component for you.
Do not compile header files (e.g. .h, .hcc). If you do so it won't be recompiled unless you delete the created .gch file.

The End