Course Overview
The Structure of a Package

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, January 18, 2013

Chris Hartman

Department of Computer Science

University of Alaska Fairbanks

cmhartman@alaska.edu

Based on material by Glenn Chappell

Course Overview
CS 311 in the CompSci & CompEng Programs

CS 311 has a dual role:

- It serves as “C.S. III”.
  - CS 201 → CS 202 → CS 311
- It introduces theoretical **computer science** (as opposed to programming, software engineering, etc.):
  - Data Structures
    - Representing data.
  - Algorithms
    - Dealing with data, accomplishing tasks.
  - Analysis of Algorithms
    - How good is an algorithm?
  - Efficiency
    - Making our programs run quickly.

Course Overview
Goals

After taking this class, you should:

- Have experience writing and documenting high-quality code.
- Understand how to write robust code with proper error handling.
- Be able to perform basic analyses of algorithmic efficiency, including use of “big-*O*” notation.
- Be familiar with various standard algorithms, including those for searching and sorting.
- Understand what data abstraction is, and how it relates to software design.
- Be familiar with standard data structures, including their implementations and relevant trade-offs.

Course Overview
Topics

The following topics will be covered, *roughly* in order:

- Advanced C++
- Software Engineering Concepts
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting
- Data Abstraction
- Basic Abstract Data Types & Data Structures:
  - Smart Arrays & Strings
  - Linked Lists
  - Stacks & Queues
  - Trees (various types)
  - Priority Queues
  - Tables
- Other, as time permits: graph algorithms, external methods.

Goal: Practical generic containers

A **container** is a data structure holding multiple items, usually all the same type.

A **generic** container is one that can hold objects of client-specified type.

Course Overview
Two Themes

Two themes will pop up over & over again this semester:

- **Robustness**
  - *Robust* code is code that always behaves reasonably, no matter what input it is given.
  - Not the same as reliability. *Reliable* code always does what you tell it to do. (But building reliable systems generally requires robust components.)
- **Scalability**
  - Code, an algorithm, or a technique is *scalable* if it works well with increasingly large problems.
  - Speed is the major issue here, of course.

Course Overview
Language

We will achieve our goals, in part, by doing an in depth study of a particular programming language, along with its standard libraries.

- We will study ANSI C++ (1998 standard, updated in 2003, augmented by Library Technical Report 1, TR1) and the Standard Template Library.
  - Any reasonably recent C++ compiler should be fine.
  - You may use the Chapman 103 Lab, which has C++ compilers available.
- We will *encourage* use of C++11, the new 2011 standard, but not require it.

Course Overview
Generic Programming

An important topic in this class is **generic programming**.

- We write code so that it can handle arbitrary data types.
- We separate algorithms from data.
- Generic programming can make fancy data structures much more practical.
- In C++, generic programming is facilitated primarily by **templates**.

Compare with **object-oriented programming**, covered in CS 202, which is facilitated primarily by inheritance and virtual dispatch.

Unit Overview
Advanced C++ & Software Engineering Concepts

We now begin a unit on advanced C++ programming and software engineering concepts.

- Some of this will be review from CS 201/202.

Major Topics

- Advanced C++
  - The structure of a package
  - Parameter passing
  - Operator overloading
  - Silently written & called functions
  - Pointers & dynamic allocation
  - Managing resources in a class
  - Templates
  - Containers & iterators
  - Error handling
  - Introduction to exceptions
  - Introduction to Linked Lists
- Software Engineering Concepts
  - Abstraction
  - Invariants
  - Testing
  - Some principles

These two will be covered concurrently.

Later in the semester we will cover other advanced C++ topics:

- Exception safety
- The C++ Standard Template Library

The Structure of a Package
Basics [1/2]

By a **package** we mean a program, library, or similar collection of code & related files that is distributed as a unit.

A package may include:

- Documentation.
- Source code.
- Makefiles or other information on how to build.
- Pre-compiled files (libraries or executables).
- Data (images, etc.)

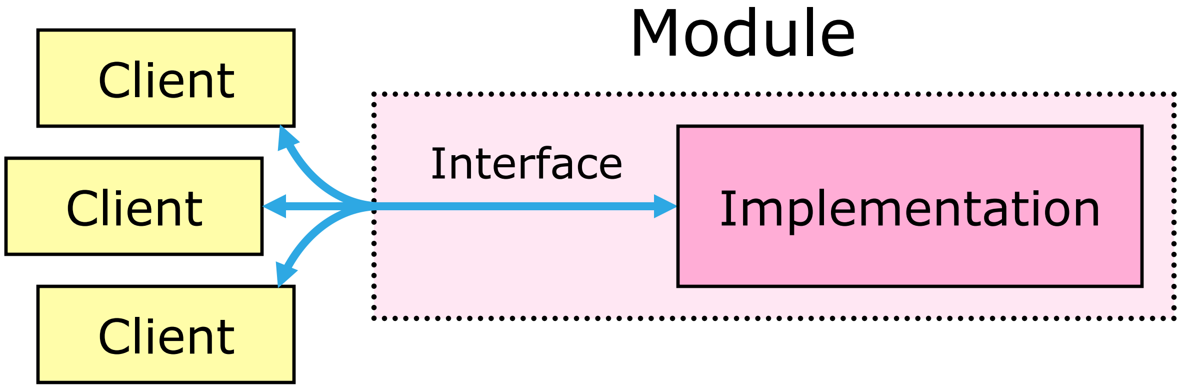
In this class:

- We require API documentation.
    - It will generally be written as comments in the code, not separate files.
    - More on this in a couple of days.
- Files should be able to be compiled in the "normal" manner.
    - Package files automatically generated by your favorite IDE should work.
- Nothing is precompiled.
- In short: Just give me the source, and put the doc's in it.

The Structure of a Package
Basics [2/2]

“**Module**” is a general term for a smaller, self-contained collection of code: a function, class, etc.

- A **client** of a module is code that uses the module.
- The **interface** of a module is how clients deal with it.
- The **implementation** is how the module is written internally.



Note: Here, a *client* is code; a **user** is a person.

The Structure of a Package
Types [1/3]

The **type** of a value or variable indicates the set of values it can take on and the operations available on it.

Examples of C++ types:

- **Simple types**: int, double, char, long, etc.
- Pointers: pointer-to-int (int *), etc.
- Array-of-double …

```
int n;   // Declaration of variable of type int
3        // Value of type int
(3+n)    // Expression whose value has type int
```

A **type conversion** takes a value and returns a value of another type.

```
int n = 3;
double d1 = n;          // Implicit conversion int to double
double d2 = double(n);  // Two explicit conversions
double d3 = static_cast<double>(n)  // None of these change n!
```

The Structure of a Package
Types [2/3]

In C++, we can define our own types in three ways:

- Using **class** (or **struct**).

```
class Foo {       // Define a type called Foo
    …
};
Foo * myFooPtr;   // Declare variable of type pointer-to-Foo
```

- Using **typedef** to create an "alias" for an existing type.
  - Idea: Write the code as if you are declaring a variable of that type, and put "**typedef**" before it.

```
typedef Foo FooArrTen[10];  // Array type
FooArrTen aa;               // Same effect as Foo aa[10]
```

- Using **enum** to create new integer constants.

```
enum WeekDay { sun = 1, mon, tue, wed, thu, fri, sat };
                         // Named enum type
WeekDay myBirthday = mon;
enum { MIN_SIZE = 20 };  // Unnamed enum type
int k = MIN_SIZE;
```

C++11 Feature
"using"

- New use of keyword `using`
  - Instead of

    ```
    typedef Foo FooArrTen[10];  // Array type
    ```

  - You may now write

    ```
    using FooArrTen = Foo[10];  // Array type
    ```

- Strongly typed enumerations
  - From Wikipedia article on C++11
  - "In C++03, enumerations are not type-safe. They are effectively integers, even when the enumeration types are distinct. This allows the comparison between two enum values of different enumeration types. " etc.

The Structure of a Package
Types [3/3]

Class members can be:

- Variables (data members).
- Functions (member functions, methods).
- Types (member types).

```
class MyContainer {
public:
    typedef double value_type;
    class MemberClass {
        …
    };
    …
};
MyContainer::value_type x;  // x is a double
MyContainer::MemberClass y;
```

The Structure of a Package
Identifiers [1/3]

Identifiers (representing functions, types, variables, etc.) in C++ have **declarations** and **definitions**.

- A *declaration* simply says that the item exists, and indicates the type.
- A *definition*, as the word suggests, defines the item.

In C++, functions and classes can have **many declarations**, but should only have **one definition**.

The Structure of a Package
Identifiers [2/3]

Function declaration (also called a “**prototype**”):

```
int theFunc(int & x);
```

Function [declaration and] definition:

```
int theFunc(int & x)
{
    x += 10;
}
```

The Structure of a Package
Identifiers [3/3]

Class declaration:

```
class TheClass;
```

Class [declaration and] definition:

```
class TheClass {
private:
    void f1(int & x);
    void f2(int & x)
    { x *= 3; }
};
```

Member function declaration

Member function [declaration and] definition

Member function [declaration and] definition outside the class definition:

```
void TheClass::f1(int & x)
{ x *= 2; }
```

Just before the *name* of the member function!

The Structure of a Package
File Conventions [1/4]

We have been looking at things that are required by the specification of the C++ language.

In addition, there are a number of **conventions**.

- A *convention* is an agreed-on practice.

One convention is that C++ code comes in two kinds of files: **header** files and **source** files.

- *Header* files are generally intended to be included by other files.
  - Header files often contain class definitions with only declarations of the members.
  - Names of header files usually end with the suffix “`.h`”.
    - Other possibilities include “`.hpp`”.
    - Most *standard* headers have no suffix (e.g., “`iostream`”).
- *Source* files are generally intended to be **compiled separately**.
  - Source files often contain mostly function definitions.
  - Names of source files end with suffixes like “`.cpp`”, “`.cxx`”, “`.c++`”, “`.C`”, “`.cc`”, etc.

The Structure of a Package
File Conventions [2/4]

Header (**`myclass.h`**) defines the **interface** for **`MyClass`**.

```
#ifndef MYCLASS_H  // This avoids multiple inclusion
#define MYCLASS_H

class MyClass {
public:
    int f(int & x);
};

#endif //#ifndef MYCLASS_H
```

Always base this on the **name of the file** (so that two files never share the same constant).

Source (**`myclass.cpp`**) usually has most of the **implementation** of **`MyClass`**.

```
#include "myclass.h"  // Note the quotes!

int MyClass::f(int & x)
{ x *= 14; }
```

`#include <` … **`>`** for system headers.

`#include "` … **`"`** for other headers.

The Structure of a Package
File Conventions [3/4]

Here is some other file (**`whatever.cpp`**) that uses **`MyClass`**

- That is, it is a **client** of **`MyClass`**.

```
#include "myclass.h"

void foo()
{
    MyClass q;
    int i = 3;
    q.f(i);
}
```

Now, **`whatever.cpp`** and **`myclass.cpp`** can be compiled separately.

- Changes in the **implementation** of **`MyClass`** (in **`myclass.cpp`**) do not require re-compilation of **`whatever.cpp`**, as long as the **interface** (in **`myclass.h`**) remains unchanged.

The Structure of a Package File Conventions [4/4]

The header file includes:

- **Declarations** of everything in the public interface.
  - Functions.
  - Classes.
  - Other types (**`typedef`**, **`enum`**).
  - Global variables.
- **Definitions** of publicly available classes.
  - Members are *usually* not defined here, but most of them can be, if you want.
  - Why “usually not”?
    - To facilitate **separate compilation** (thus reducing compile time).
    - To hide implementation details from clients.
  - We might define short, simple member functions here.
- **Definitions** of things that cannot be compiled separately.
  - Functions declared inline.
  - Templates.

The Structure of a Package
Wrap-Up

Some concepts to know:

- Interface & Implementation
- Client
- User
- Type
- Simple type
- Type conversion (implicit & explicit)
- Identifier
- Declaration & Definition
- Function prototype
- Header & Source
  - Put things in the right place!
- Convention
- Separate Compilation