CPSC 427a: Object-Oriented Programming

Michael J. Fischer

(with thanks to Ewa Syta for help with the slides)

Lecture 14
October 20, 2011

More on Course Goals

Demo: Stopwatch

Demo: Hangman Game
- Game Rules

More on Course Goals

Low-level details

- C++ is a large and complicated language with many quirks and detailed rules.
- One goal of this course is for you to learn how to deal effectively with a complex system where it is not feasible to know everything about it before beginning to use it.
- Low-level details tend to be easy to find in the documentation once you know what to look for.
- What's important to learn is the overall roadmap of the language and where to look to find out more.

Example picky detail

- If you do not supply a constructor for a class, C++ automatically generates a null default constructor for you, that is, one that takes no parameters and does nothing.
- If you do define a constructor, the default constructor is *not* generated. If you want it, you then need to explicitly define it, e.g.,

```
MyClass() {}
```

- What if you didn't know this and assumed the default constructor was pre-defined? The compiler would give you an error comment about it not being defined, and you would be started on the track of trying to figure out why.

Efficient use of resources

Efficiency is concerned with making good use of available resources:

- Time (how fast a program works)
- Memory (how much memory the program requires)
- Other resources that are scarce and relatively costly to create:
  - Network connections (TCP sockets)
  - Database connections

Strategy for improving efficiency: Reuse and recycle. Maintain a pool of currently unused objects and reuse rather than recreate when possible.

In the case of memory blocks, this pool is often called a **free list**.

Efficiency measurement

A first step to improving efficiency is to know how the resources are being used.

Measuring resource usage is not always easy.

The next demo is concerned with measuring execution time.

Demo: Stopwatch

How to measure run time of a program

- There is no standard procedure in C++ for accurately measuring time.
- Time measurement depends on the software clocks provided by your computer and operating system.
- Clocks advance in discrete clicks called **jiffies**. A jiffy on the Zoo linux machines is one millisecond (0.001 seconds) long.
- Even if the clock is 100% accurate, the measured time can be off by as much as one jiffy.
- Hence, times shorter than tens of milliseconds cannot be directly measured with much accuracy using the standard software clock.

High resolution clocks

- Linux also provides high resolution clocks based on CPU timers.
- High resolution clocks are useful to the operating system for task scheduling and timeouts.
- They are also available to the user for higher-precision time measurements.
- Be aware that reading the clock involves a kernel call that takes a certain amount of time. This itself may limit the accuracy of timing measurements, even when the clock resolution is sufficiently high for the desired accuracy.
- See `man 7 time` for more information about linux clocks.

Measuring time in real systems

- Measuring code efficiency in real systems is challenging. Many factors can influence the results that are hard to control.
  - Other process running on the same machine.
  - Time spent in the OS moving data on and off disks.
  - Memory caching behavior.
- Lacking a controlled laboratory environment, one can still take measures to improve accuracy of the tests:
  - Do some tests to determine what factors seem to have a sizable effect on the run time, e.g., the first run of a program is likely to be slower than subsequent runs because of caching.
  - Run the same test several times to get a feeling for the variance of results.
  - Make sure the optimizer isn't optimizing away code that you think is being executed.

Realtime measurements

`StopWatch` is a class I wrote for measuring realtime performance of code.

It emulates a stopwatch with 3 buttons: `reset`, `start`, and `stop`.

At any time, the watch displays the cumulative time that the stopwatch has been running.

(See demo.)

HirezTime class

HirezTime is a wrapper class for the system-specific functions to read the clock.

It hides the details of the underlying time representation and provides a simple interface for reading, computing, and printing times and time intervals.

HirezTime objects are intended to be copied rather than pointed at, and to behave like other numeric types.

Versions of HirezTime

There are two versions:

12-StopWatch (Linux/Unix/Darwin) Function `gettimeofday()` returns the clock in a `struct timeval`, which consists of two `long int`s representing seconds and microseconds. The resolution of the clock is system-dependent, typically 1 millisecond.

12-StopWatch-hirez (Linux only) Function `clock_gettime()` returns the clock in a `struct timespec`, which consists of two `long int`s representing seconds and nanoseconds. The resolution of the clock is system-dependent and can be obtained with the `clock_getres()` function.

`HirezTime` structure

- In C++, `struct T` and `class T` are very similar. In both cases, `T` becomes a new type name.
- `struct` members are public by default.
  `class` members are private by default.
- `HirezTime` is derived from `struct timeval` or `struct timespec`, depending on the version.
- It uses `protected` derivation to hide the underlying representation.
- It presents two interfaces to the world:
  1. The normal public interface treats `HirezTime` as an opaque object.
  2. A class derived from it can access the fields of the underlying `timespec`/`timeval`.

Printing a HirezTime number

Something seemingly simple like printing HirezTime values is not so simple. Naively, one might write:

```
cout << t.tv_sec << "." << t.tv_usec;
```

where tv_sec and tv_usec are the seconds and microseconds fields of a timeval structure.

If t represents 2 seconds and 27 microseconds, then what would print is 2.27, not the correct 2.000027.

The class contains a print function that fixes this problem.

StopWatch class

StopWatch contains five member variables to record

- Whether the watch is running or not.
- The cumulative run time to point when last stopped.
- The most recent start and stop times.

All functions are inline to minimize inaccuracies of measurement due to the overhead withing the stopwatch code itself.

Casting a StopWatch to a HirezTime

An operator extension defines a cast for reading the cumulative time from a stop watch:

```
operator HirezTime() const { return cumSpan; }
```

Thus, if `sw` is a `StopWatch` instance,

```
cout << sw;
```

will print `sw.cumSpan` using `sw.print()`.

Why it works

This works because `operator<<()` is not defined for righthand operands of type `StopWatch` but it is defined for `HirezTime`.

The compiler then **coerces** `sw` to something that is acceptable to the `<<` operator.

Because `operator HirezTime()` is defined for class `StopWatch`, the compiler will invoke it to obtain a `HirezTime` object, for which `<<` is defined.

Note that a similar coercion happens when one writes

```
    if(!in) {...}
```

to test if an `istream` object `in` is open for reading. Here, the `istream` object is coerced to a `bool` because `operator bool()` is defined inside the streams package.

Demo: Hangman Game

Game Rules

Hangman game

Well-known letter-guessing game.

Start with a hidden *puzzle word*.

Player guesses a letter.

- If letter appears in puzzle word, matching letters are uncovered.
- If letter does not appear, it is shown in list of bad guesses.

Player wins when puzzle word is uncovered.

Player loses after 7 bad guesses