# CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 13
March 3, 2016

## Smart Pointer Demo

## Dangling pointers

Pointers can be used to permit object sharing from different contexts.

One can have a single object of some type `T` with many pointers in different contexts that all point to that object.

## Problems with shared objects

If the different contexts have different lifetimes, the problem is to know when it is safe to delete the object.

It can be difficult to know when an object should be deleted. Failure to delete an object will cause memory leaks.

If the object is deleted while there are still points pointing to it, then those pointers become invalid. We call these dangling pointers.

Failure to delete or premature deletion of objects are common sources of errors in C++.

## Avoiding dangling pointers

There are several ways to avoid dangling pointers.

1. Have a top-level manager whose lifetime exceeds that of all of the pointers take responsibility for deleting the objects.
2. Use a garbage collection. (This is java's approach.)
3. Use reference counts. That is, keep track somehow of the number of outstanding pointers to an object. When the last pointer is deleted, then the object is deleted at that time.

## Modern C++ Smart Pointers

Modern C++ has three kinds of **smart pointers**. These are objects that act very much like raw pointers, but they take responsibility for managing the objects they point at and deleting them when appropriate.

- `shared_ptr`
- `weak_ptr`
- `unique_ptr`

We will discuss them later in the course. For now, we present a much-simplified version of shared pointer so that you can see the basic mechanism that underlies all of the various kinds of shared pointers.

## Smart pointers

We define a class `SPtr` of reference-counted pointer-like objects.

An `SPtr` should act like a pointer to a `T`.

This means if `sp` is an `SPtr`, then `*sp` is a `T&`.

We need a way to create a smart pointer and to create copies of them.

Demo `13-SmartPointer` illustrates how this can be done.

## 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.

## Clocks and Time Measurement

## 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.