Simple Class Example (cont.)
Testing

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, January 30, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
`cmhartman@alaska.edu`

Unit Overview
Advanced C++ & Software Engineering Concepts

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

Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- Testing
- Some principles

Review
Software Engineering Concepts: Invariants

An **invariant** is a condition that is always true at a particular point in an algorithm.

Special kinds

- **Precondition**. An invariant at the beginning of a function. The responsibility for making sure the preconditions are true rests with the calling code.
  - What must be true for the function to execute properly.
- **Postcondition**. An invariant at the end of a function. Tells what services the function has performed for the caller.
  - Describe the function’s effect using statements about objects & values.
- **Class invariant**. An invariant that holds whenever an object of the class exists, and execution is not in the middle of a public member function call.
  - Statements about data members that indicate what it means for an object to be valid or usable.

Simple Class Example
Write It!

TO DO

- Write a simple class that stores and handles a **time of day**, in seconds.
  - Call it "**`TimeSec`**".
  - Give it reasonable ctors, etc.
    - Can we use silently written functions?
      - *Yes, for the Big Three.*
      - *We will write our own default ctor.*
  - Give it reasonable operators.
    - Like what?
      - *Pre & post* **`++`**, **`--`**.
      - *Equality & inequality:* **`==`**, **`!=`**.
      - *Stream insertion:* **`<<`**.

Simple Class Example continued
Write Some More

Last time, we wrote partof Timesec.h. Today we’ll complete this.

- Write the member functions and associated global operators for class **`TimeSec`**.
- Apply the software engineering concepts just discussed.

Simple Class Example
Notes [1/2]

Note 1: External interface does not dictate internal implementation (although it certainly influences it).

- Class **`TimeSec`** deals with the outside world in terms of hours, minutes, and seconds. However, it has only one data member, which counts seconds.

Note 2: Avoid duplication of code.

- Look at the two **`operator++`** functions. We could have put the incrementing code into both of them, but we did not.
- Why is this a good thing?

Simple Class Example Notes [2/2]

Note 3: There are three ways to deal with the possibility of invalid parameters.

Responsibility for dealing with the problem lies with the code executed …

- Insist that the parameters be valid. ← … **before** the function.
  - Use a precondition.
- Allow invalid parameter values, but then fix them. ← … **in** the function.
- If invalid parameter values are passed, signal the client code that there is a problem. ← … **after** the function.
  - We will discuss this further when we get to "Error Handling".

Look at the three-parameter constructor. Which solution was used there?

Software Engineering Concepts: Testing
A Tragic Story [1/4]

Suppose you are writing a software package for a customer.

- The project requires the writing of four functions.

```
double foo(int n);    // gives ipsillic tormorosity of n
void foofoo(int n);   // like foo, only different
int bar(int n);       // like foofoo, only more different
char barbar(int n);   // like bar; much differenter
```

So, you get to work. You start by writing function **`foo`** …

Software Engineering Concepts: Testing
A Tragic Story [2/4]

… after a huge amount of effort, the deadline arrives. But you are not done. However, you do have three of the four functions written. Here is what you have.

```
double foo(int n)
{
    [amazingly clever code here]
}

void foofoo(int n)
{
    [stunningly brilliant code here]
}

int bar(int n)
{
    [heart-breakingly high-quality code here]
}

// Note to self: write function barbar.
```

Software Engineering Concepts: Testing
A Tragic Story [3/4]

You meet with the customer. You explain that you are not done. The customer is a bit annoyed, of course, but he knows that schedule overruns happen in every business.

So, he asks, “Well, what *have* you finished? What can it do?”

Unfortunately, you do not have all the function prototypes in place. Thus your unfinished package, when combined with the code that is supposed to use it, *does not even compile*, much less actually do anything.

You tell the customer, “Um, actually, it can’t do anything at all.”

“Do you want to see my beautiful code?” you ask.

“No,” replies the customer, through clenched teeth.

The customer storms off and screams at your boss, who confronts you and says you had better have something good in a week. You solemnly assure them that this will happen.

You go back to work …

Software Engineering Concepts: Testing
A Tragic Story [4/4]

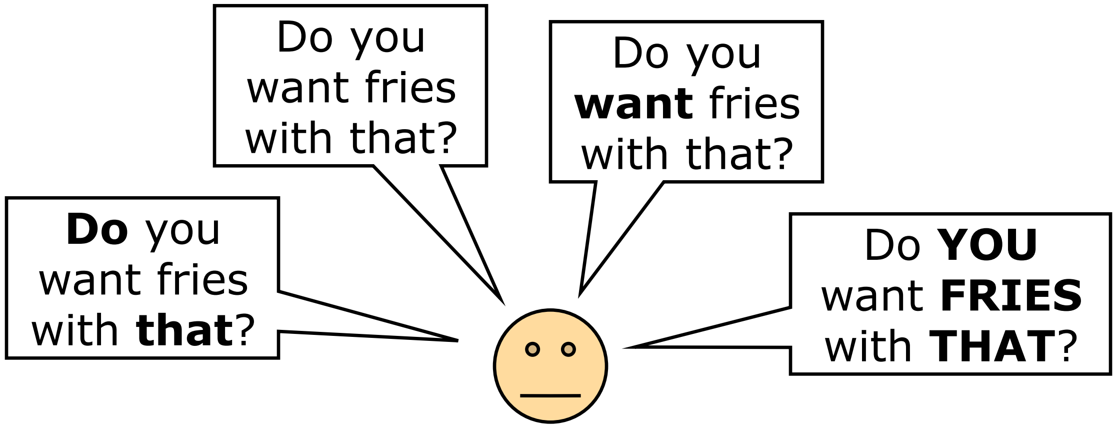
… and you write a do-nothing function **`barbar`**, just to get things to compile.

However, when you do this, you realize that, since you have never done a proper compile of the full package, you have never really done a proper test of the first three functions.

Now that you *can* test them, you find that they are full of bugs.

Alas, you now know that you have been far too optimistic; nothing worthwhile is going to get written in the required week.

You begin practicing your lines for your exciting new career:


Do you want fries with **that**?
Do you want fries with that?
Do you **want** fries with that?
Do **YOU** want **FRIES** with **THAT**?

Software Engineering Concepts: Testing Lessons

Observations

- Code that does not compile is worthless *to a customer*, even if it is "nearly done".
- It *might* not be worth anything *to anyone*; **you can't tell**, because …
- Code that does not compile cannot be tested, and so it *might* be much farther from being done than you suspect.
- Testing is what uncovers bugs.

Conclusion

- First priority: **Get your code to compile**, so that it can be tested.

A Revised Development Process

- Step 1. Write dummy versions of all required modules.
  - Make sure the code **compiles**.
- Step 2. Fix every bug you can find.
  - "Not having any code in the function body" is a bug.
  - Write notes to yourself in the code.
  - Make sure the code **works**.
- Step 3. Put the code into finished form.
  - Make it pretty, well commented/documented, and in line with coding standards.
  - Many comments can be based on notes to yourself.
  - Make sure the code is **finished**.

Software Engineering Concepts: Testing
Try Again [1/3]

Suppose you had used this revised development process earlier.

Step 1. Write dummy versions of all required modules.

```
double foo(int n)   // gives ipsillic tormorosity of n
{}  // WRITE THIS FUNCTION!!!
void foofoo(int n)  // like foo, only different
{}  // WRITE THIS FUNCTION!!!
int bar(int n)      // like foofoo, only more different
{}  // WRITE THIS FUNCTION!!!
char barbar(int n)  // like bar; much differenter
{}  // WRITE THIS FUNCTION!!!
```

Does it compile?

- No. My compiler says **`foo`**, **`bar`**, **`barbar`** must each return a value.

Software Engineering Concepts: Testing
Try Again [2/3]

Continuing Step 1.

Add dummy **`return`** statements.

```
double foo(int n)   // gives ipsillic tormorosity of n
{ return 1.; }    // WRITE THIS FUNCTION!!!
void foofoo(int n)  // like foo, only different
{}                // WRITE THIS FUNCTION!!!
int bar(int n)      // like foofoo, only more different
{ return 1; }     // WRITE THIS FUNCTION!!!
char barbar(int n)  // like bar; much differenter
{ return 'A'; }   // WRITE THIS FUNCTION!!!
```

Does it compile?

- Yes. Step 1 is finished.

Software Engineering Concepts: Testing
Try Again [3/3]

Step 2. Fix every bug you can find.

You begin testing the code. Obviously, it performs very poorly. But you begin writing and fixing. And running the code. So when something does not work, *you know it*. When you figure something out, you make a note to yourself about it.

As before, the deadline arrives, but the code is not finished yet.

You meet with the customer. “The project is not finished,” you say, “but **here is what it can do**.”

You estimate how long it will take to finish the code.

You can make this estimate with confidence, because you have a list of tests that do not pass; you know exactly what needs to be done.

Software Engineering Concepts: Testing
Development Methodologies

Software-development methodologies often include standards for how code should be tested.

- In particular, see, “Test-Driven Development” and “Agile Programming”

Many people recommend *writing your tests first*.

- Each time you add new feature, you first write tests (which should fail), then you make the tests pass.
- When the finished test program runs without flagging problems, Step 2 is done. Pretty up the code, and it is finished.

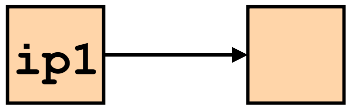
We will use a variation on this in the assignments in this class.

- I will provide the (finished) test program.
- However, when you turn in your assignment, I act as the customer; I do not want to see code that does not compile.

Pointers & Dynamic Allocation
Dynamic Allocation [1/2]

The **`new`** operator allocates and constructs a value of a given type and returns a pointer to it.

```
ip1 = new int;
```



We can add constructor parameters.

```
ip1 = new int(2);
```

When we do dynamic allocation, we must deallocate using **`delete`** on a pointer (any pointer) to the dynamic value.

```
delete ip1;  // destroys the int; does not affect ip1
```

Do not depend on the destructor of the pointer to do this. The destructor of a pointer does **nothing**.

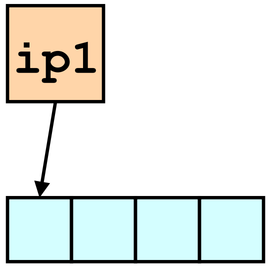
- When dynamic memory is never deallocated, we have a **memory leak**.

Pointers & Dynamic Allocation
Dynamic Allocation [2/2]

The **`new`** and **`delete`** operators also have array forms.

```
ip1 = new int[4];

delete [] ip1;
```



Notes

- You cannot specify constructor parameters in the array version of **`new`**. Array items are always default constructed.
- Do not mix array & non-array versions. Use the proper form of **`delete`** for each **`new`**.

Software Engineering Concepts: Some Principles
Coupling

Coupling: the degree of dependence between two modules.

- **Loose** coupling: little dependence. Can modify one module without breaking (and thus being required to modify) others.
- **Tight** coupling: a lot of dependence. Changing one thing breaks other things. System is **brittle**: easy to break.
- *Some* coupling is unavoidable. But less (loose) is better.

Software Engineering Concepts: Some Principles
DRY

DRY: Don't Repeat Yourself

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
  - Possibly originated with A. Hunt & D. Thomas in *The Pragmatic Programmer* (1999).

Software Engineering Concepts: Some Principles
SRP & Cohesion

SRP: Single Responsibility Principle

- Every module should have exactly one well-defined responsibility.
  - Originated with R. C. Martin, in the context of OOP, early 2000s.
- A module that follows SRP is said to be **cohesive**.

Preview: SRP/cohesion helps with error handling.

- **Failure** happens, even in good software.
- Ideally, when failing, restore to original state.
- Suppose a function has two responsibilities that involve changing data, and the *second* one fails.

Managing Resources in a Class
Preliminaries — Exceptions

When a function encounters an error condition, this often needs to be communicated to the caller (or the caller's caller, or the caller's caller's caller, or …).

One way to do this is by **throwing an exception**.

- This causes control to pass to the appropriate handler.
- When an exception is thrown, a function can exit in the middle, despite the lack of a **`return`** statement.

We will discuss exceptions in a few days, and again later in the class. For now, be aware that:

- Throwing an exception can result in a function being exited just about anywhere.
  - In particular, if function **`foo`** calls function **`bar`**, and function **`bar`** throws, then function **`foo`** can then exit.
- When a function exits, whether by a normal **`return`** or by throwing an exception, destructors of all automatic objects are called.

Managing Resources in a Class
Problem & Solution — The Problem

What is “scary” about code like this?

```
void scaryFn(int size)
{
    int * buffer = new int[size];
    if (func1(buffer))
    {
        delete [] buffer;
        return;
    }
    if (func2(buffer))
    {
        delete [] buffer;
        return;
    }
    func3(buffer);
    delete [] buffer;
}
```

Function **`scaryFn`** has 3 exit points.

- The buffer must be freed in each.
- Otherwise, it will **never** be freed. This would be a **memory leak**.

If we alter the code in this function, it is easy to create a memory leak accidentally.

In fact, there may be other exit points, if one of the 3 functions called ever throws an exception.

- In that case, function **`scaryFn`** has a memory leak already.

Now, imagine a different scenario: some memory is allocated and freed in different functions.

- What if it might be freed in *one of several* different functions?
- Memory leaks become hard to avoid.

Managing Resources in a Class
Problem & Solution — About Destructors

We want to solve this problem.

First, consider the following facts:

- The destructor of an **automatic** (local non-static) object is called when it goes out of scope.
    - This is true no matter whether the block of code is exited via **`return`**, **`break`** (for loops), **`goto`** (ick!), hitting the end of the block of code, or an exception.
- The destructor of a **static** (global, local, or member) object is called when the program ends.
- The destructor of a non-static **member** object is called when the object of which it is a member is destroyed.

In short, execution of destructors is something we can depend on, except for:

- **Dynamic** objects (those created with **`new`**).

Therefore …

Managing Resources in a Class
Problem & Solution — A Solution: RAII

Solution

- Each dynamic object, or block of dynamically allocated memory, is managed by some other object.
- In the **destructor** of the managing object:
  - The dynamic object is destroyed.
  - The dynamically allocated memory is freed.

Results

- Destructors always get called.
- Dynamically allocated memory is always freed.

This programming idiom is, somewhat misleadingly, called **Resource Acquisition Is Initialization** (**RAII**).

- The name would seem to refer to allocation in the constructor. Actually, we may choose not to do that, but we always **deallocate in the destructor**.
- So "RAII" is not terribly good terminology, but it is standard.

Managing Resources in a Class
Ownership — Idea

In general (RAII or not), to avoid memory leaks, we need to be careful about "who" (that is, what module) is responsible for freeing a block of memory or destroying a dynamic object.

- Whatever has this responsibility is said to **own** the memory/object.

For example, a **function** can own memory.

- This is what we saw in function `scaryFn`.

When we use RAII, each dynamic object (or block of memory) is owned by some other **object**.

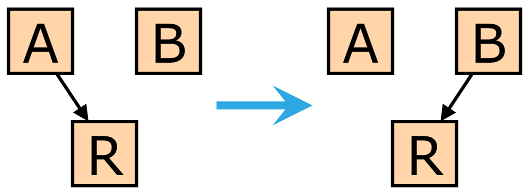
Ownership = Responsibility for Releasing

RAII = An Object Owns
(and, therefore, its destructor releases)

Managing Resources in a Class
Ownership — Transfer, Sharing

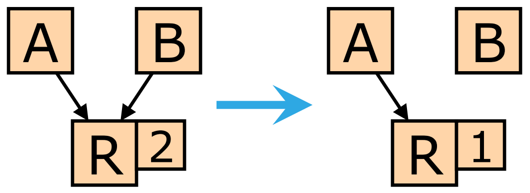
Ownership can be **transferred**.

- Think of a function that allocates an array and returns a pointer to it.
- Objects can transfer ownership, too.



Ownership can be **shared**.

- Keep track of how many owners a block has: a **reference count**.
- When a new owner is added, increment the reference count.



- When an owner relinquishes ownership, decrement the count.
- When the count his zero, deallocate.
  - "The last one to leave turns out the lights."

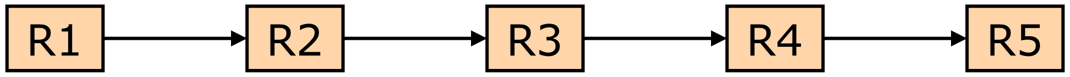
Reference-Counted "Smart Pointers"

- The C++11 standard has reference-counted pointers (`std::shared_ptr<>`).
- Newer languages often have such pointers built-in (e.g., Python).

Managing Resources in a Class
Ownership — Chaining

The idea of ownership can make complex situations easy to handle.

Suppose object R1 owns object R2, which owns object R3, which owns object R4, which owns object R5.



- When R1 goes away, the other four must also, or we have a leak.
- However, each object only needs to destroy the **one** object it owns.
- Thus, each object can have a one-line destructor.

More Generally

- An object only needs to release resources that it directly owns.
- If those resources manage other resources, that is their business.
- RAII makes all this happen automatically.

Managing Resources in a Class
Ownership — Invariants

Ownership is an important **invariant**.

- When ownership is transferred to a function, it is a precondition of the function.
- When a function transfers ownership upon exiting, it is a postcondition of the function.
- When we use RAII, ownership is a class invariant.

In each case, we need to document the ownership.

- Usually as a precondition, postcondition, or class invariant.

The only time we do not need to document ownership is when it begins and ends within a single function.

- But it still might be a good idea (think about **`scaryFn`**).