Software Engineering Concepts: Some Principles
Managing Resources in a Class

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, September 16, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

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.

Review
Software Engineering Concepts: Testing

Recommended development process:

- Step 1. Get the code to **compile**.
  - Write dummy versions of all modules.
- Step 2. Get the code to **work**.
  - Fill in blank spots. Test & fix bugs. ("Not written yet" is a bugs.)
- Step 3. Get the code **finished**.
  - Comment, finalize the documentation, make everything pretty.

The first step is getting code that compiles!

- Code that compiles can be tested.
- We find problems by testing.
- "Working" means you can test it and not find any problems.

An idea: test first.

- When adding a feature, first write tests (which should fail), then write the code to make them pass.

In this class, I write the tests.

Review
Simple Class Example

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

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

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

← … **before** the function.

← … **in** the function.

← … **after** the function.

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 not 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.

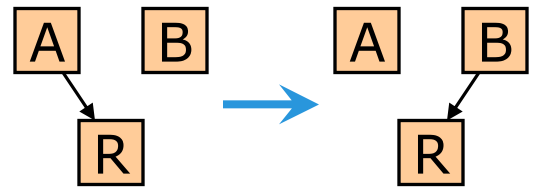
- This is what we saw in function `scaryFn`.

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

Ownership = Responsibility
for Releasing

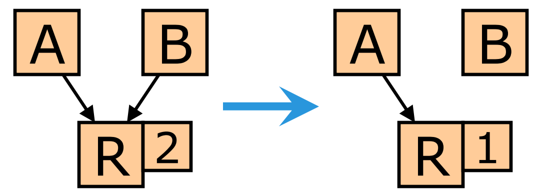
RAII = An Object Owns
(and, therefore, its destructor releases)

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."

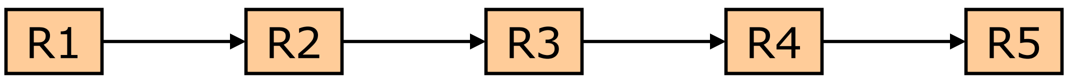
Reference-Counted "Smart Pointers"

- A good implementation is in the Boost library (**`Boost::shared_ptr`**).
- The next C++ standard is likely to have reference-counted pointers.
- 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.

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**).

Managing Resources in a Class
Generalizing — Other Kinds of Resources [1/2]

The concepts of ownership and RAII can be applied to resources other than dynamically allocated memory.

- Files
  - Suppose a file is open. Who closes it?
- Concurrent Programming Support
  - Semaphores, etc., must be allocated. Who frees them?
- Other System Resources
  - Desktop windows.
  - Network connections.
  - Etc.
- Anything that needs to be cleaned up (somehow) when you are done with it.

Managing Resources in a Class
Generalizing — Other Kinds of Resources [2/2]

When we clean up a resource and formally relinquish control over it, we are **releasing** the resource.

- Freeing dynamically allocated memory and destroying objects in it.
- Flushing and closing a file.
- Closing and destroying a desktop window.
- Terminating a network connection.
- Etc.

When a resource is never released, we have a **resource leak**.

The **owner** of a resource is responsible for releasing it.

RAII

- A resource is owned by an object.
- Therefore, the object’s destructor releases the resource.

Ownership of any resource is an important invariant.

- Document it, unless it begins and ends within a single function.

Managing Resources in a Class
Generalizing — Example

RAII is used by C++ standard streams classes, to manage open files.

Example:

```
bool handleInput(const std::string & filename)
{
    std::ifstream inFile(filename.c_str());  // open the file
    if (!inFile) return false;
    for (int i = 0; i < 10; ++i)
    {
        int inValue;
        inFile >> inValue;
        if (!inFile) return false;
        processInput(inValue);
    }
    return true;
}
```

Q: Where is the file closed?

A: In the dctor of **inFile**.

That is, **here** or **here** or **here**.

Or *possibly* **here**, if **processInput** can throw an exception.

Managing Resources in a Class
Generalizing — Law of the Big Three

Recall "The Big Three"

- Copy ctor
- Copy assignment
- Dctor

The Law of the Big Three

- If you need to declare one of these, then you probably need to declare all of them.
- Note: When you eliminate them, you are still declaring them.

Resource ownership is the usual (only?) reason for declaring the Big Three.

Managing Resources in a Class
An RAII Class — Starting

Rather than have an object *directly* manage every resource it deals with, it is sometimes convenient to write small wrapper classes that use RAII.

Let’s write a simple RAII class that owns a dynamic integer array.

- Call it “**`IntArray`**”.
- What is the **absolute minimum functionality** that such a class must have, to be useful in improving a function like our **`scaryFn`**?
- Rewrite **`scaryFn`** to use the new class.

Managing Resources in a Class
An RAII Class — `std::size_t`

- The C++ standard header `<cstdlib>` defines some types that help in writing system-independent code:
  - Type `std::size_t`
    - An unsigned integer type big enough to hold the size of any object.
    - "`t`" for "type".
  - Type `std::ptrdiff_t`
    - Like `size_t`, only signed (can be negative). Gets its name from the fact that it can hold the difference between two pointers.
- The header *probably* defines `std::size_t` as `unsigned long`. But using `size_t` is better than using `unsigned long`, since it works on all systems. It also gives a hint what the value is for.
- Indices for arrays of arbitrarily large size, object sizes, etc. should usually be one of these types.

```
#include <cstdlib>
IntArray(std::size_t size)  // size = # of ints in array
```

Managing Resources in a Class
An RAII Class — **`typedef`** & Member Types

New types defined using **`typedef`** can help clarify the purpose of a variable.

```
int breakfastTime = 8;

typedef int HourOfDay;
HourOfDay breakfastTime = 8;
```

We can define **member types** that tell client code what types to use with a class.

```
class IntArray {
public:
    typedef std::size_t size_type;

    IntArray(size_type size) // size = # of ints in array
```

Type used for sizes when dealing with class **`IntArray`**

Public member types can be used outside of the class just as data and function members can. Here, we can refer to "**`IntArray::size_type`**".

Managing Resources in a Class
An RAII Class — Constness

In general, we want to be able to change items in an **`IntArray`**. But we want a **`const IntArray`** to manage an array whose items cannot be changed.

```
IntArray nc(20);      // non-const
cout << nc[1];        // Legal
nc[1] = 2;            // Legal

const IntArray c(20); // const
cout << c[1];         // Legal
c[1] = 2;             // SHOULD NOT COMPILE
```

How can we do this?

- Answer: have two versions of the bracket operator. One non-const, one const. They are identical, except for the types involved.
- This idea is common, when dealing with access to data managed by an object.

```
int & operator[](size_type index)
{ return arrayPtr_[index]; }
const int & operator[](size_type index) const
{ return arrayPtr_[index]; }
```

Managing Resources in a Class
An RAII Class — The **explicit** Keyword

Implicit type conversion: silent function call, converts between types.

- Some are built-in, like the implicit conversion from **int** to **double**.

```
void foo(double x);
foo(3);  // 3: int, not double; implicit conversion is done.
```

1-parameter ctor: used for implicit conversions unless declared **explicit**.

```
class IntArray {
public:
    explicit IntArray(size_type size);
```

- If the above were not **explicit**, then the compiler could convert an **int** to an **IntArray**. For example "**3**" could become an **IntArray** with 3 uninitialized items. (Ick!)

We declare most 1-parameter ctors **explicit**. But not the copy constructor; this would disallow passing by value.

Managing Resources in a Class
An RAII Class — Write It

TO DO

- Write class **`IntArray`**.
- Rewrite function **`scaryFn`** to use it.

*Partially done. See **`intarray.h`**, on the web page.*

Managing Resources in a Class
TO BE CONTINUED …

Managing Resources in a Class will be continued next time.