

Managing Resources in a Class

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, February 6, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Advanced C++

Major Topics

- ✓ • The structure of a package.
- ✓ • Parameter passing & “const”.
- ✓ • Operator overloading.
- ✓ • Silently written and silently called functions.
- ✓ • Pointers and dynamic allocation.
 - Managing resources in a class.
 - Templates.
 - Containers & iterators.
 - Error handling.
 - Introduction to exceptions.

Review

Simple Class Example — Notes

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

Note 2: Avoid duplication of code.

Note 3: There are three ways to deal with possible errors involving 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 that there is a problem.

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, destructors of all automatic objects are called.

Managing Resources in a Class

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

A 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.

So ...

Managing Resources in a Class

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.

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

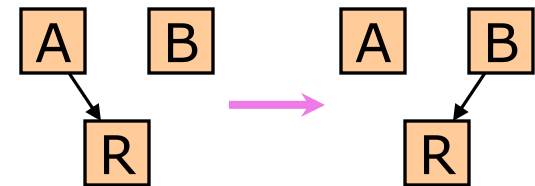
RAII = An Object Owns
(and, therefore, the destructor releases)

Managing Resources in a Class

Ownership — Transfer, Sharing

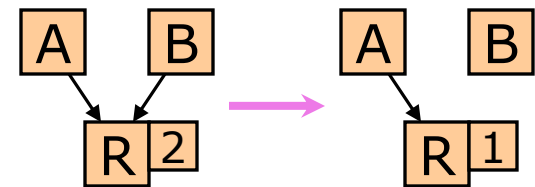
Ownership can be **transferred**.

- 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 is zero, deallocate.
 - “The last one to leave turns out the lights.”
- Reference-Counted “Smart Pointers”
 - These are tricky to write correctly.
 - A good implementation is in the Boost library (`Boost::shared_ptr`).
 - The next C++ standard is likely to have reference-counted pointers.
 - Some languages (e.g., Python) have such pointers built-in.



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.

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.
- 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 the C++ Standard Library streams classes.

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: You declare them when you eliminate them.

Resource ownership is the usual reason for declaring the Big Three.

Managing Resources in a Class

Writing an RAII Class — Starting

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

Let's write a simple RAII class the owns an 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`? (Think: complete, minimal interface).

The following options were mentioned in class.

- *Constructor*
 - Given size, allocates array.
 - OR, given pointer to previously allocated array, takes ownership.
- *Destructor*
- *Way of accessing data in array*
 - Bracket operator
 - OR, "get" function, returning pointer.
- Rewrite `scaryFn` to use the new class.

We implemented these options.

Managing Resources in a Class

Writing 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.

Array indices, 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

Writing an RAII Class — `typedef` & Member Types

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

```
int breakfastTime = 8;
```

← What does "8" mean?

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

← This is clearer (?).

We can define **member types** that tell the client 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

Writing 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

Writing an RAII Class — The `explicit` Keyword

Implicit type conversions are silent function calls that convert one type to another.

- Some of these are built-in. For example, there is an conversion from `int` to `double`.

```
void foo(double x);
```

```
foo(3); // 3 is an int, not a double. An implicit conversion is done.
```

One-parameter constructors can be silently called as implicit type conversions unless they are declared “explicit”.

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

- If the above were not explicit, then the compiler could convert an integer to an `IntArray`. For example “3” could become an `IntArray` with 3 uninitialized items. (Ick!)

We generally declare most one-parameter constructors explicit. But not the copy constructor; this would disallow passing by value.

Managing Resources in a Class

Writing an RAII Class — Do It

TO DO

- Write class `IntArray`.
- Rewrite function `scaryFn`.

*Done. See `intarray.h`,
on the web page.*

← Next time.