Managing Resources in a Class continued
Containers & Iterators

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, September 4, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- ✓ Invisible functions I
- ✓ Integer types
- (part) Managing resources in a class
- Containers & iterators
- Invisible functions II
- Error handling
- Using exceptions
- A little about Linked Lists

Major Topics: S.E. Concepts

- ✓ Invariants
- ✓ Testing
- Abstraction

Review

Review
Software Engineering Concepts: Testing

A development process:

- Step 1. Make sure the code **compiles**.
  - Write dummy versions of all components.
- Step 2. Make sure the code **works**.
  - Fill in blank spots. Test & fix bugs. (*No code in function body* is a bug.)
  - In this step, the code should always compile.
- Step 3. Make sure the code is **finished**.
  - Finalize comments & documentation. Make sure everything is pretty.
  - In this step, the code should always work.

The first step is getting code that compiles!

- Code that compiles can be tested. Bugs can be found and fixed.
- "Working" means you can test it thoroughly and find no problems.

Review
Integer Types

For integer values, use `int` for not-very-large numbers, or …

Use a type that reflects your intent. For example:

- `std::size_t` for object sizes & array indices.
- `std::ptrdiff_t` for similar values that may be negative.
- `std::uint_fast64_t` for an unsigned 64-or-more-bit integer.
- There are useful member types, like `vector<Foo>::size_type`.

We can make our own member types.

```
class FooList {
public:
    using size_type = size_t;
    using value_type = Foo;
```

Client code can now use `FooList::size_type`.

Review
Managing Resources in a Class [1/2]

Exceptions may cause a function to exit, even where there is no return. Destructors of automatic objects are still called.

Dynamically allocated memory & objects need clean-up when we are done with them.

- If we never **deallocate**: there is a **memory leak**.

Own memory/object = be responsible for releasing (deallocating).

Ownership = Responsibility for Releasing

Prevent memory leaks with **RAII**.

- Memory/object is owned by an object.
- Therefore, its destructor releases—if this has not been done yet.
- Define or =delete each of the Big Five in an RAII class.

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

Review
Managing Resources in a Class [2/2]

We (mostly) wrote an RAII class to manage a dynamic `int` array.

See `intarray.h`.
See `intarray_main.cpp`
for a simple main program.

Minimal functionality:

- Initialize array (ctor takes size & allocates).
- Access array (bracket operator).
- Clean up array (dctor).

Some relevant ideas:

- Use `std::size_t` (`<cstddef>`) for sizes & indices.
- Member types can be helpful (e.g., `size_type`, `value_type`).
- Tricky constness issues come up when we write a bracket operator.
- `explicit`: prevent a one-parameter ctor from being used to do implicit type conversions.

Managing Resources in a Class

continued

Managing Resources in a Class
An RAII Class — MORE CODE

TO DO

- Finish class `IntArray`.
  - Constructor from size (explicit).
  - Destructor.
  - Bracket operator (both const & non-const).
  - Member types `size_type`, `value_type`.
- Rewrite function `scaryFn` to use `IntArray`.

Done. See `intarray.h`.

See the next slide.

Managing Resources in a Class
An RAII Class — Usage in a Function

Original `scaryFn`

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

New `scaryFn`, using `IntArray`

```
void scaryFn(size_t size)
{
    IntArray buffer(size);
    if (func1(&buffer[0]))
        return;
    if (func2(buffer))
        return;
    func3(&buffer[0]);
}
```

This line supposes that `func2` has been rewritten to take an `IntArray` parameter.

The parameter cannot be passed by value, because `IntArray` has no copy/move ctors.

Managing Resources in a Class
An RAII Class — Usage in a Class

Class with an Array Member

```
class HasArray {
public:
    HasArray(size_t size)
        :_theArray(new int[size])
    {}

    ~HasArray()
    { delete [] _theArray; }

    …

    void out(size_t index) const
    { cout << _theArray[index]; }

private:
    int * _theArray;
};
```

Same idea, using `IntArray`

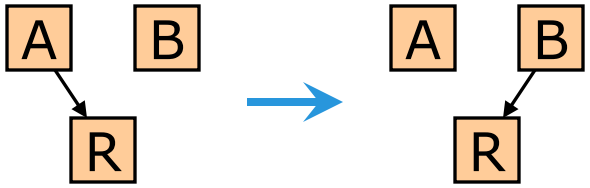
```
class HasArray {
public:
    HasArray(size_t size)
        :_theArray(size)
    {}

    // Auto-generate dctor
    ~HasArray() = default;

    …

    void out(size_t index) const
    { cout << _theArray[index]; }

private:
    IntArray _theArray;
};
```

Same

Managing Resources in a Class
More on Ownership — Transfer, Sharing

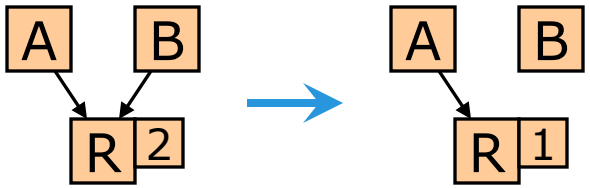
Ownership can be **transferred**.

- 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

- Since the 2011 standard, the C++ Standard Library has had a reference-counted smart-pointer template: `std::shared_ptr<T>`

Managing Resources in a Class
More on Ownership — Chaining

Ownership can make some complex situations easy to handle.

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

- When R1 goes away, the other two 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 typically only needs to release resources it directly owns.
- If those resources manage other resources, that is their business.
- RAII makes all this happen automatically.

Note. Applying this idea to very long chains can result in problems with excessive *recursion depth*. More on this later.

Managing Resources in a Class
Generalizing Ownership [1/2]

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

- An open file (who is responsible for closing it?)
- Network connections.
- Or anything else that needs clean-up when we are done with it.

Acquire a resource: get access and control.

Release a resource: clean it up and relinquish control.

So:

- If a resource is never released, then we have a **resource leak.**
- The **owner** of a resource is responsible for releasing it.
- **RAII**: an object owns a resource. Its destructor releases.
- Ownership of a resource is an important invariant.
  - Document it, unless it begins and ends within a single function—and maybe even then, too.
- Direct resource ownership is the usual reason to define/=delete the Big Five.

Managing Resources in a Class
Generalizing Ownership [2/2]

RAII is used by standard stream classes, to manage open files.

```
bool handleInput(const std::string & filename)
{
    std::ifstream inFile(filename);
    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`.

Strictly speaking, not here, since this exit is taken if the file could not be opened. But if you guessed this spot, then your heart is in the right place. ☺

Here or **here** …

… or *possibly* **here**, if `processInput` may throw an exception.

Managing Resources in a Class
Notes — Circular References

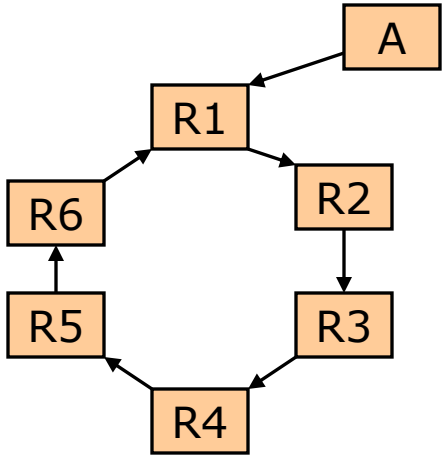
The idea of ownership breaks down in one situation: when there are **circular references**.

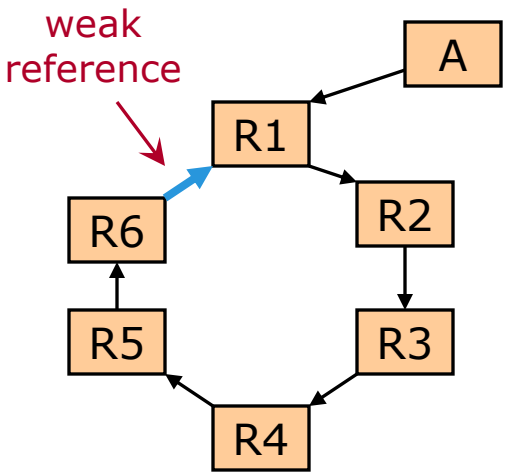
- If A is released, then R1 .. R6 are not released. There is a resource leak.

One solution: weak references.

- A **weak reference** is a non-owning reference ("reference" in a general sense; *maybe* a pointer) to a resource.
- Weak references can be dangerous; they may result in a resource being released too early, if you are not careful.

Another solution is a **garbage collector** that checks for circular references. However, this requires knowing the structure of objects.





Would we write and use `IntArray` in practice?

- Unlikely.
- First, `IntArray` does not offer complete management of its resource: it cannot copy or move.
- Second, the C++ Standard Library already includes smarter RAII array class templates (`std::vector`, `std::array`, and `std::basic_string`), as well as simpler ownership-only smart pointer classes (`std::unique_ptr` and `std::shared_ptr`).

However, we could certainly apply the ideas of ownership and RAII in real-world projects.

In situations where no existing resource-management classes fit our needs, we might need to write one or more, based on the principles covered here.

Containers & Iterators

Containers — Generic Containers

A **container** is a data structure that can hold multiple items, usually all of the same type.

A **generic container** is a container that can hold items of a client-specified type.

One kind of generic container: a C++ built-in array.

```
MyType myArray[8];
```

Other generic container types are in the C++ Standard Library.

In particular, the **Standard Template Library** (**STL**), contains templates for many data structures that can hold arbitrary types, as well as algorithms that can deal with arbitrary types.

Containers — Troubles with C++ Built-In Arrays

STL containers are necessary, because C++ built-in arrays have very few operations defined on them.

- There is no resizing and no "size" member function—no member functions at all, actually.
- There is no copy or assignment. When a built-in array is passed by value, it **decays** to a pointer to its first item.

```
int a[10];
func(a);
func(&a[0]); // Same as above
// func cannot tell the size of the array it receives
```

Containers — std::vector [1/3]

We would prefer a container type that is *first-class*.

- A type is **first-class** if it can be tossed around with the ease of something like `int` (for example, new values can be created at runtime, they can be passed to and returned from functions, and they can be stored in containers).

One generic container found in the STL: `std::vector`.

- `vector` is a first-class array.
- It is declared in the standard header `<vector>`.
- This is a class template, not a class.

```
vector v1;          // DOES NOT COMPILE!
vector<int> v2;     // vector of int
```

Containers — std::vector [2/3]

Like any array, `vector` has lookup by index:

```
vector<int> v3(20);  // Much like int arr[20];
cout << v3[5] << endl;
v3[19] = 7;
```

A `vector` knows how to copy itself:

```
v3 = v2;
```

A `vector` knows its size.

```
cout << v3.size() << endl;
```

Containers — std::vector [3/3]

A default-constructed `vector` has size 0. But there are other ctors.

```
vector<Blug> v4(20);  // Holds 20 items of type Blug;
                      //  all are default-constructed
vector<double> v5(55, 7.);  // Holds 55 doubles, all 7.
```

We can change the size of a `vector`:

```
v5.push_back(6.1);  // Adds new item at end, value 6.1
v5.pop_back();      // Eliminates last item
v5.resize(20);      // v5 now has size 20
```

I call `std::vector` a **smart array**.

Containers & Iterators
TO BE CONTINUED …

Containers & Iterators will be continued next time.