Managing Resources in a Class continued
Templates

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, September 18, 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
- (part) 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

DONE

Review
Software Engineering Concepts: Some Principles

Three Principles

- **Coupling**: degree of dependence between modules
  - **Loose** vs. **tight**.
  - Some is unavoidable. We like loose.
  - Tight coupling leads to **brittle** systems.
- **DRY**: Don’t Repeat Yourself
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
- **SRP**: Single Responsibility Principle
  - Every module should have exactly one well-defined responsibility.
  - Same as **cohesion**.

Review
Managing Resources in a Class [1/2]

Some **resources** need to be cleaned up when we are done with them.

- Quintessential example: dynamic objects.
- Others: files to be closed, windows to be destroyed, locks to be released, etc.
- We **acquire** a resource. Later, we **release** it.
- If we never release: **resource leak**.

Own a resource = be responsible for releasing.

- Ownership can be transferred, shared (using a **reference count**), and "chained".
- Ownership is an invariant. Document it.
- Write The Big Three when a resource is owned.

Prevent resource leaks with **RAII**

- A resource is owned by an object.
- And therefore, the **destructor** of the object releases the resource, if necessary.

Ownership = Responsibility for Releasing

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

Review
Managing Resources in a Class [2/2]

Exceptions were briefly introduced.

We wish to write an RAII class to manage a dynamic array of **`int`**s.

Minimum functionality:

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

Some relevant ideas:

- Use **`std::size_t`** (in **`<cstdlib>`**) for sizes & indices.
- **Member types** can be helpful (e.g. **`size_type`**).
- Tricky constness issues come up when we write a bracket operator.
- The **`explicit`** keyword prevents new implicit type conversions.

Managing Resources in a Class continued
An RAII Class — Write It

TO DO

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

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

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

Original **scaryFn**

```
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;
}
```

New **scaryFn**, using **IntArray**

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

Say function **func2** has been rewritten to take an **IntArray** parameter. This must be passed by reference or reference-to-const.

If, we had decided that the **IntArray** ctor was given a pointer, then we would say

```
IntArray buffer(new int[size]);
```

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

Class with an Array Member

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

    ~HasArray()
    { delete [] theArray_; }

    [ other stuff goes here ]

    void printIt(int index) const
    { cout << theArray_[index]; }

private:
    int * theArray_;
};
```

Same idea, using **`IntArray`**

```
class HasArray {
public:
    HasArray(int size)
        :theArray_(size)
    {}

    // Use compiler-generated
    //  dctor

    [ other stuff goes here ]

    void printIt(int index) const
    { cout << theArray_[index]; }

private:
    IntArray theArray_;
};
```

Same

Managing Resources in a Class
Note — Circular References

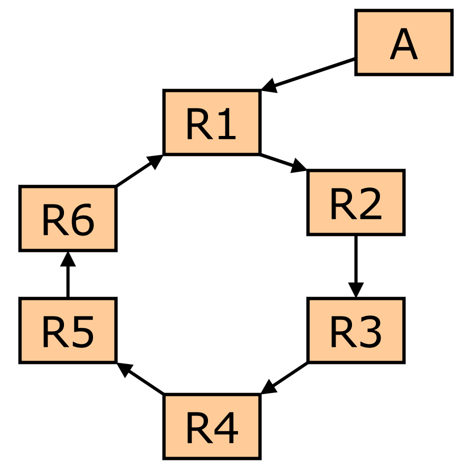
The idea of ownership breaks down in one situation: when there are **circular references**.

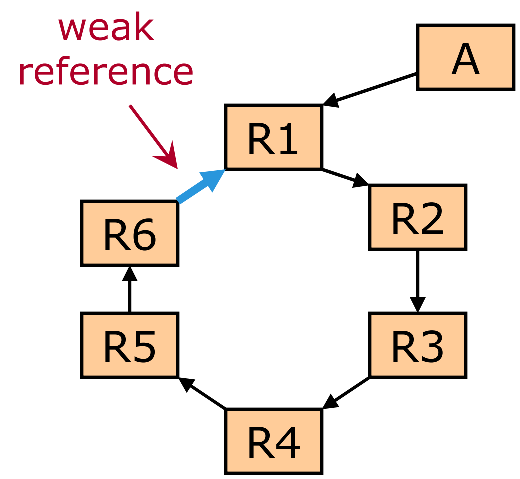
- If A is released, then R1 .. R6 are not released. There is a resource leak.

One solution: **weak references**.

- A *weak reference* is a non-owning "reference" (in the 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.





Templates
Introduction

In C++, templates are a way of writing code without specifying the types it deals with.

- Templates are the primary structure used in **generic programming**.

Templates usually cannot be separately compiled.

- Therefore, when defining templates, put everything in the header (**`.h`**) file. No source file is needed.

C++ has:

- **Function templates**
- **Class templates**

We now look at these in more detail.

Templates
Function Templates — Basics

Example function: add one to **int**

```
int plusOne(int x)
{
    return x + 1;
}
```

Example **function template**: add one to anything

- Below, “**T**” is a **template parameter**.

```
template <typename T>  // "T" is traditional; use any name you want
T plusOne(T x)  // Treat "T" as a type
{
    return x + 1;
}
```

Usage of function template

```
double d2 = plusOne(3.7);
```

Templates
Function Templates — Write One

Write a function template to convert *anything* to a string.

- Anything printable, that is.

```
#include <string>   // for std::string
#include <sstream>  // for std::ostringstream

template <typename T>
std::string toString(const T & value)
{
    std::ostringstream os;
    os << value;
    return os.str();
}
```

Templates
Class Templates — Basics

Example class: holds one **int**

```
class SingleValue {
public:
    int & val()
    { return theValue_; }
    const int & val() const
    { return theValue_; }
private:
    int theValue_;
};
```

Inside the class template definition, the template parameter **ValueType** is a type.

Example class template: holds one of anything

```
template <typename ValueType>
class SingleValue {
public:
    ValueType & val()
    { return theValue_; }
    const ValueType & val() const
    { return theValue_; }
private:
    ValueType theValue_;
};
```

Usage of class template

- Need to specify the template parameter.

```
SingleValue<double> sd;
```

Templates
Class Templates — Ctors, etc.

When you use a class template outside its own definition, specify the template parameter.

```
SingleValue<int> x;
void foo(const SingleValue<int> & y)
{ … }
```

The **name** of a ctor in a class template is the name of the class template.

- Similarly for the dctor.

Inside the definition of a class template, you may leave off template parameters when referring to the **current class**.

```
template <typename T>
class Bar {
    Bar();                              // default ctor
    Bar(const Bar & other);             // copy ctor
    Bar & operator=(const Bar & rhs);   // copy =
    ~Bar();                             // dctor
};
```

Templates
Class Templates — Write One

Write the dctor and copy ctor for this class template:

```
// class HasPointer
// Invariants:
//     myPtr_ points to a T allocated with new,
//      owned by *this.
template <typename T>
class HasPointer {
public:
    HasPointer(const HasPointer & other)
        :myPtr_(new T(*other.myPtr))
    {}
    HasPointer & operator=(const HasPointer & rhs);
    ~HasPointer()
    { delete myPtr; }
private:
    T * myPtr_;
};
```

Because of this, we *must* define the Big Three, and the copy ctor *must* do a **deep copy**.

Templates
Documenting

When you write a template with a type as a template parameter, **document** the requirements on that type.

- Include things that the compiler checks (unlike in invariants).
- In this course, put this information in a comment.

```
// cubeIt
// Returns the cube of the given number.
// Requirements on types:
//     Num must have a copy ctor and binary operator*.
// Pre: None.
// Post:
//     return == n*n*n.
template <typename Num>
Num cubeIt(Num n)
{
    return n*n*n;
}
```

What has to be true about type **`Num`** for this template to be compiled and used successfully?