

# Managing Resources in a Class, cont'd

## Templates

---

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, February 8, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

[CHAPPELLG@member.ams.org](mailto: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.
- 1/2✓ • Managing resources in a class.
  - Templates.
  - Containers & iterators.
  - Error handling.
  - Introduction to exceptions.

# Review

## Managing Resources in a Class — The Problem

---

Scary code:

```
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;
}
```

This code is scary, because it has 3 exit points, each of which must deallocate.

- If one of the 3 functions called can throw an **exception**, then there may be other exit points.

If we modify this function, it is easy to forget to handle deallocation properly. Result: a **memory leak**.

- If an exception can be thrown, then there is already a memory leak.

## Review

### Managing Resources in a Class — Resources

---

Dynamic memory is an example of a **resource**: something that needs to be cleaned up when we are done with it.

- Others: files that need to be closed, windows that need to be destroyed, locks that need to be released.
- We **acquire** a resource. When we are done, we **release** it.
- We might forget to release. Result: a **resource leak**.

## Review

# Managing Resources in a Class — A Solution

---

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

- Ownership can be transferred, shared (using a **reference count**), and “chained”.
- Resource ownership is an important invariant. Document it.
- We generally write The Big Three (copy ctor, copy assignment, dctor) when a resource is owned.

Solution: **RAII**

- “Resource Acquisition Is Initialization” [unfortunate terminology].
- In RAII, a resource is owned by an object. The object’s destructor releases the resource (if necessary).

**Ownership** = Responsibility  
for Releasing

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

## Review

# Managing Resources in a Class – Writing an RAII Class

---

We wrote an RAII class to manage an `int` array:

### `IntArray`.

- Absolute minimum functionality:  
ctor taking size, dctor, bracket op.

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

### Notes

- Use `std::size_t` (in `<cstdint>` for size, array indices).
- Member typedefs can help:  
`"typedef std::size_t size_type"`.
- When giving access to internal data (e.g., with the bracket operator), have both `const` & non-`const` versions.
- Declare a 1-parameter ctor `"explicit"` if it should not be used to do an implicit type conversion.

Next, we rewrite `scaryFn` to use `IntArray`.

# Managing Resources in a Class, cont'd

## Writing 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, instead, we had decided that an `IntArray` was constructed from a pointer, then we would say

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

# Managing Resources in a Class, cont'd

## Writing 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
    // dtor

    [ other stuff goes here ]

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

private:
    IntArray theArray_;
};
```

Same!

## Managing Resources in a Class, cont'd

### Circular References

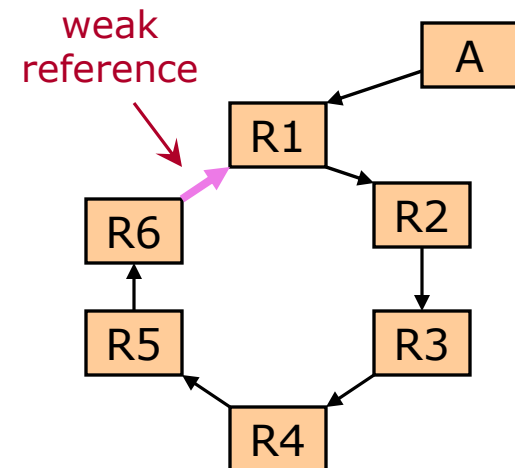
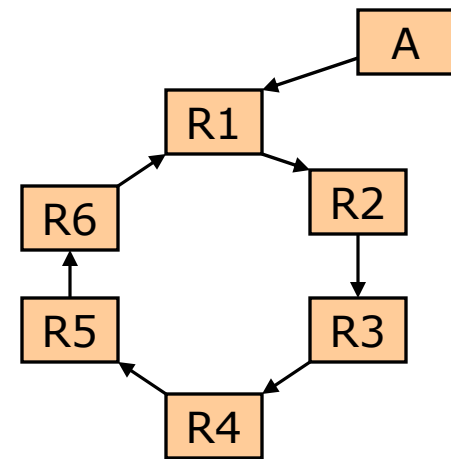
---

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 aren't careful.
- *I wish we knew of a better solution to this problem.*



# 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 — Usage

---

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

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

Inside a definition of a template, you may use a template parameter (that is a type) **any way you can use a type**.

- In particular, you may use it as a template parameter for another template.

```
template <typename SomeType>  
void foofoo(const SingleValue<SomeType> & y)  
{ ... }
```

# Templates

## Class Templates — Member Function Definitions

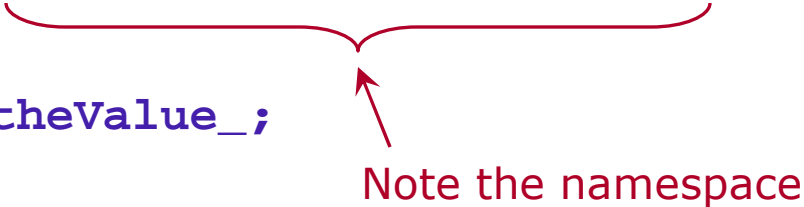
---

When defining a member function of a class template outside the class (template) declaration, treat it as a function template.

```
// class template definition
template <typename ValueType>
class SingleValue {
    ValueType & val(); // member function declaration
};
```

```
// member function definition
template <typename ValueType>
ValueType & SingleValue<ValueType>::val()
{
    return theValue_;
}

```



Note the namespace

# Templates

## Class Templates — Ctors, etc.

---

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

- Similarly for the dtor.

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

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

Outside the class (template) definition, specify the template parameter(s).

```
template <typename T>
MyClass<T>::MyClass(const MyClass<T> & rhs) // copy ctor
{ ... }
```

# Templates

## Class Templates — Write One

---

Define (outside the class definition) the 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);
    HasPointer & operator=(const HasPointer & rhs);
    ~HasPointer();
private:
    T * myPtr_;
};

template <typename T>
HasPointer<T>::HasPointer(const HasPointer<T> & other)
    :myPtr_(new T(*other.myPtr_))
{ }
```

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

**In practice**, you will probably define all member functions inside the class definition (no source file, remember?).