

Templates, cont'd

Containers & Iterators

Error Handling

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, February 11, 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.
- 1/2✓ • Templates.
 - Containers & iterators.
 - Error handling.
 - Introduction to exceptions.

Review

Templates — Function Templates

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

Here is a **function template** to convert anything printable 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();
}
```

Review

Templates — Class Templates

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, cont'd

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 class, 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?

Containers & Iterators

Introduction — Generic Containers

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

- Sometimes people talk about a “container” holding a single item.

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

One kind of generic container is: an array.

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

Other generic container types are part of the C++ Standard Library.

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

Containers & Iterators

Introduction — Kinds of Data

When we deal with containers (and things that look like containers [think “data abstraction”]) the following broad categories of data are important:

- **Random Access**
 - Random-access data can be dealt with in any order. We can efficiently skip from one item to any other item in the data set.
- **Sequential Access**
 - Sequential-access data is data that can only be dealt with (or only dealt with efficiently) in order. We begin with some item, then proceed to the next, etc.
 - Sequential access data may be **one-way**, accessible only in forwards order. Or it may be **two-way**, accessible in both forwards and backwards order.

Containers & Iterators

Introduction — What is Wrong with Arrays?

In C++, arrays are not **first-class types**.

- They have no copy or assignment operations.
- When an 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 does not know size of array
```

- In fact, array types have **no member functions at all**, not even ctors.

```
MyClass arr[7];
```

- The above does not call a `MyClass[]` constructor, since there is no such thing. Instead, it makes 7 calls to the `MyClass` default constructor.

In general (not just in C++), arrays perform poorly when doing some operations: for example, inserting a new item in the middle.

Containers & Iterators

Smart Arrays & `std::vector` — What are They?

A **smart array**:

- Works pretty much like a regular array, except ...
- It is a first-class type.
 - It can be copied, etc.
- It knows its size.
- It can change its size, maybe?

The C++ STL includes a smart array: `std::vector`.

- Declared in the standard header `<vector>`.
- Is a **class template**, not a class.

```
std::vector v1; // DOES NOT COMPILE!
```

```
std::vector<int> v2; // vector of ints
```

Containers & Iterators

Smart Arrays & `std::vector` — Using `vector` [1/2]

A `vector` works like an array:

```
std::vector<int> v3(20); // Like int array[20];
cout << v3[5] << endl;
v3[19] = 7;
```

However it is a first-class type:

```
void func1(std::vector<int> x)
{ ... }
```

```
v3 = v2;
func1(v2);
```

- Note: This is legal; however, for efficiency we usually pass `vectors` and other objects by reference-to-const or by reference.

Containers & Iterators

Smart Arrays & `std::vector` — Using `vector` [2/2]

A `vector` knows its **size**.

```
std::vector<int> v4;  
cout << v4.size() << endl;
```

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

```
std::vector<Blug> v5(20); // Holds 20 default-constructed Blugs  
std::vector<double> v6(30, 7.2); // Holds 30 doubles, all 7.2
```

We can **change the size** of a `vector`:

```
Blug b;  
v5.push_back(b); // Adds new item at the end; sets it to b  
v5.pop_back(); // Eliminates last item  
v5.resize(10); // v5 now has size 10
```

Containers & Iterators

Loops — Types of Loops

You are familiar with **counter-controlled loops**:

- Also called “**for**” loops.

```
for (int i = 0; i < 100; ++i)
    cout << i * i << endl;
```

... and **condition-controlled loops**:

- Also called “**while**” loops. But in C++ we *can* write these using the “for” construct.

```
while (!infile.eof())
{
    readFrom(infile);
    if (!infile) break;
}
```

Now we look at a third kind: **iterator-controlled loops**:

- Also called “**for-each**” loops.

```
int array[7];
for (int * p = array; p != array+7; ++p)
    *p = 6;
```

Containers & Iterators

Loops — Iterator-Controlled Loops with `vector`

Iterator-Controlled Loops

- With an array:

```
int array[7];  
for (int * p = array; p != array+7; ++p)  
    *p = 6;
```

- With a `std::vector`:

```
std::vector<int> v(7);  
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)  
    *it = 6;
```

- As above, but using `typedef`:

```
typedef std::vector<int> IVec;  
IVec v(7);  
for (IVec::iterator it = v.begin(); it != v.end(); ++it)  
    *it = 6;
```

Points to
first item

Points to
one-past last item

Containers & Iterators

Iterators — What are They?

“Iterator” is a slightly vague term.

- Generally, an **iterator** is a variable that acts like a pointer, particularly as pointers are used in the following:

```
for (int * p = array; p != array+7; ++p)
    *p = 6;
```

Iterators:

- Refer to items in containers.
 - Or they act like it, anyway.
 - Think “data abstraction”.
- Usually allow (at least) rudimentary pointer-arithmetic-style operations and manipulation.
 - Default ctor, Big Three, equality tests (==, !=), increment (++), dereference (*).
- Usually **do not involve ownership** of what they point to.

Containers & Iterators

Iterators — Examples

As we have seen, **pointers** can be used as iterators.
STL containers have associated **iterator types**.

```
std::vector<int>::iterator iter;           // Like normal pointer
std::vector<int>::const_iterator citer;    // Like pointer-to-const
*iter = 3;                                 // Okay
*citer = 3;                             // DOES NOT COMPILE!
cout << *citer;                           // Okay
```

An iterator can be a **wrapper** around data, to make it look like a container.

```
#include <iterator>
std::ostream_iterator<int> myCoolNewIterator(std::cout, "\n");
```

- Now the following two lines do the same thing:

```
std::cout << 3 << "\n";
*myCoolNewIterator++ = 3; // Same as above
```

Containers & Iterators

Iterators — ... and Generic Algorithms

Why do we want to have many kinds of iterators?

- This allows us to access different kinds of data using the same interface.
- Write an algorithm to take an iterator, and it can deal with any kind of data.
- This is part of what is called “**generic programming**”.

Example

- Algorithm `std::copy`, defined in `<algorithm>`, copies one range to another.

```
#include <algorithm>
int arr1[20];
int arr2[20];
std::copy(arr1, arr1+20, arr2);           // Copy arr1 to arr2.

std::vector<int> v(20);
std::copy(v.begin(), v.end(), arr2);     // Copy v to arr2.

std::copy(v.begin(), v.end(), myCoolNewIterator);
    // Print the items in v, one on each line!
```

Containers & Iterators

Iterators — Specifying Ranges [1/2]

One thing we often do with iterators is to specify a **range** of items.

- This is nearly always done in the same way:
 - Two iterators.
 - The first points to the first item.
 - The second points to just past the last item.
- We will see that this is a very convenient way to do things.

Examples

```
#include <algorithm>
```

```
int arr3[100];
```

```
std::sort(arr+27, arr+90); // Sort arr3[27..89].
```

```
std::sort(v.begin(), v.end()); // Sort all of vector v.
```

Containers & Iterators

Iterators — Specifying Ranges [2/2]

More Examples

```
#include <algorithm>
int arr3[100], arr4[30];

std::copy(arr3+4, arr3+17, arr4+10);
    // Copy arr3[4..16] to arr4, starting at arr4[10].
    // That is, copy arr3[4..16] to arr4[10..22].

void printInt(int n)
{ cout << n << endl; }

std::for_each(v.begin(), v.end(), printInt);
    // Print the items in v, each on a separate line.
    // Note that std::for_each is a function template,
    // not a language flow-of-control structure.
```

Containers & Iterators

Iterators — ... and Kinds of Data

Operations available on an iterator match the underlying data.

- Iterators for one-way sequential-access data have the ++ operation.
 - Such an iterator is called a **forward iterator** (example of an **iterator category**).

```
++forwardIterator;
```

- Iterators for two-way sequential-access data also have the -- operation.

```
++bidirectionalIterator;
```

```
--bidirectionalIterator;
```

- Iterators for random-access data have all the pointer arithmetic operations.

```
++randomAccessIter;
```

```
--randomAccessIter;
```

```
randomAccessIter += 7;
```

```
cout << randomAccessIter[5];
```

```
std::size_t distance = randomAccessIter2 - randomAccessIter1;
```

Error Handling

Error Conditions

An **error condition** (often simply “error”) is a condition occurring during runtime that cannot be handled by the normal flow of execution.

An error condition is not the same as a bug in the code.

- So we are not referring to compilation errors.
- But *some* error conditions are caused by bugs.
- However, in our discussion of error handling, we will usually assume that our code is properly written.

An error condition does not mean that the user did something wrong.

- Although *some* error conditions are caused by user mistakes.

Example

- Suppose we have a function `copyFile`, which opens a file, reads its contents into a buffer, and writes them to another file.
- Function `copyFile` is called in order to read a file on a device that is accessed via a network.
- Halfway through reading the file, the network goes down.
- The function is supposed to read the file. This is now impossible. The normal flow of execution cannot handle this. We have an error condition.

Error Handling

Dealing with Error Conditions

Sometimes we can **prevent errors**:

- Write a precondition that requires the caller to keep a certain problem from happening.
- Example: Insisting on a non-zero parameter, to prevent a division-by-zero error condition.

Sometimes we can **contain errors**, by handling them ourselves:

- If something does not work, fix it.
- Example: To run a fast algorithm, we need a large buffer. Memory is low, and we cannot allocate the buffer. So we run a slower algorithm that needs no buffer.

But sometimes we can do neither of these ...

Then we must **signal the client**.

- Rule of thumb: Signal the client when the function is unable to fulfill its postconditions.
- Example: The earlier file-reading example.