Templates
Containers & Iterators

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, February 6, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

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
- ✓ 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
Managing Resources in a Class

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)

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)


    HasPointer & operator=(const HasPointer & rhs);
    ~HasPointer()

private:
    T * myPtr_;
};
```

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

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.

```
// squareIt
// Returns the square of the given number.
// Requirements on types:
//
// Pre: None.
// Post:
//     return == n*n.
template <typename Num>
Num squareIt(Num n)
{
    return n*n;
}
```

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

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.

```
// squareIt
// Returns the square of the given number.
// Requirements on types:
//     Num must have a copy ctor and binary operator*.
// Pre: None.
// Post:
//     return == n*n.
template <typename Num>
Num squareIt(Num n)
{
    return 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, or even holding no items.

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?

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
```

C++ arrays have few operations defined on them. In fact, C++ array types have **no member functions at all**, not even ctors.

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

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

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
```

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

A **`vector`** works much 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: The above is legal; however, for efficiency we usually pass **`vector`**s and other objects by reference-to-const or 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** ("for" loops):

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

Do something a certain number of times, or iterate over a sequence of numbers.

… and **condition-controlled loops**: ("while" loops):

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

Iterate until something happens.

Now we look at a third kind: **iterator-controlled loops**: ("**for-each**" loops"):

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

Iterate over the items in a container or range ("for each item …").

Iterator-Controlled Loops

- With an array:

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

Points to first item

Points to one-past last item

- 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;
```

Containers & Iterators
Iterator Basics — 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 (`*`).
- **Do not involve ownership** of what they point to.

Containers & Iterators
Iterator Basics — 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
Iterator Basics — 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

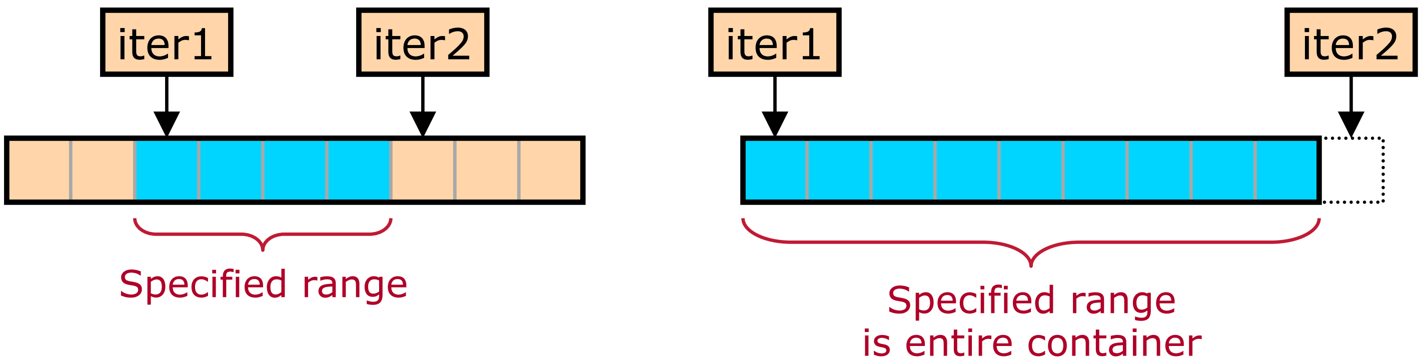
- Algorithm **`std::copy`**, defined in **`<algorithm>`**, copies one range to another.

```
#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
Iterator Basics — Specifying Ranges [1/2]

To specify a range, we use two iterators:

- An iterator to the first item in the range.
- An iterator to just past the last item in the range.



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.
```

Iterator Basics — 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
Iterator Basics — 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.
  - These are **bidirectional iterators**.

```
++bidirectionalIterator;
--bidirectionalIterator;
```

- Iterators for random-access data have all the pointer arithmetic operations.
  - These are **random-access iterators**.

```
++randomAccessIter;
--randomAccessIter;
randomAccessIter += 7;
cout << randomAccessIter[5];
std::ptrdiff_t dist = randomAccessIter2 - randomAccessIter1;
```

Containers & Iterators
Wrap-Up: Three STL Algorithms to Know

Be familiar with the following STL algorithms (all in **<algorithm>**):

- Copying: **std::copy**

```
std::copy(v.begin(), v.end(), v2.begin());
    // Copy items in v to v2 (which must have space!)
```

- For-each loop: **std::for_each**

```
std::for_each(v.begin(), v.end(), myFunc);
    // Call myFunc on each item in v
```

- Sorting: **std::sort**

```
std::sort(v.begin(), v.end());
    // Arrange items in v in ascending order
```

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 Possible 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.

Handle the error **before** the function

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.

Handle the error **during** the function

But sometimes we can do neither of these …

Then we must **signal the client code**.

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

Handle the error **after** the function

Error Handling
Goals and Guarantees (Preview)

In situations in which the client code might need to be informed of errors, there a three things we would *like* to happen:

- First, we want to be sure that an error will not "mess up" our program. It should be able to continue to run, and, later, to terminate properly. Objects must still be usable. Also, resources should not be leaked.
- In addition, we would like it if, when we want to perform an operation, either the operation completes successfully with no errors, or, if there is an error, no change is made to the program's data.
- Best of all, we would like it the client never needs to be informed of an error at all.

Later in the class, we will formalize these as "safety guarantees".

- The first idea above is the fundamental standard that all quality code *must* meet. We will call it the "Basic Guarantee".
- The second is preferred, although sometimes we do not achieve it. We will call it the "Strong Guarantee".
- The third is mostly wishful thinking. Errors happen, and sometimes the client needs to be informed. But in special cases (often involving "finishing things up"), we may need to insure that the client never needs to be informed of an error. We will call this the "No-Throw Guarantee", or "No-Fail Guarantee".

Error Handling
Flagging Errors

When we cannot prevent or contain an error, we must signal the client code. **How?**

Method 1: Returning an error code

- Here we indicate an error by our return value (or a reference parameter).
- The old "C" I/O library uses this method:

```
int c = getc(myFile);
if (c == EOF)
    printf("End of file\n");
```

Method 2: Setting a flag to be checked by a separate error-checking function

- Here the caller uses some other function to check whether there was an error.
- C++ file streams use this method by default:

```
char c;
myFileStream >> c;
if (myFileStream.eof())
    cout << "End of file" << endl;
```

Error Handling
Need for Another Method

Return codes and separate error-checking functions are both fine methods for flagging errors, but they do have problems.

- They can be difficult to use in places where a value cannot be returned, or an error condition cannot be checked for.
  - Constructors & destructors. Also bracket operator, etc.
  - In the middle of an expression.
  - When you call someone else's function, and that calls your function, which needs to signal an error condition.
    - Call-back functions, templates, etc.
- They can lead to complicated code.
  - A function calls a function, which calls a function … and an error occurs. To handle the error, we have to back out of all of these. Lots of `if`'s.

To deal with these problems, a third method was developed.

Method 3: **Throwing an exception**

- Exceptions are available in many languages (C++, Java, Python, Ruby, Javascript, etc.), and are generally associated with OOP.
- Shortly, we will look at exception handling in C++.

Summary
Error Handling

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

- Not necessarily a bug or a user mistake.
- Example: Could not read file.

Three ways to deal with a *possible* error condition in a function:

before — **Prevention**
Client code must prevent the error (precondition).

during — **Containment**
Fix the problem inside the function.

after — **Signal the Client Code**
Idea: When we cannot fulfill our postconditions.

We **like** these two, but they *might* not be feasible

Methods for signaling an error condition to the client code:

- Return an error code
- Set a flag, checked by a separate function
- Throw an **exception**