CS 151
Advanced C++ Programming

Module 13 – Exceptions, Templates, and the STL (Standard Template Library)

Chris Merrill
Computer Science Dept.
cmerrill@miracosta.edu

Agenda

- Discuss Final and Research Projects
- Exceptions and Exception Handling (Chapter 16)
- Function and Class Templates
- The Standard Template Library
- Review of homework and lab
- Quiz next week up through this Module

Topics

16.1 Exceptions

16.2 Function Templates

16.3 Class Templates

16.4 Class Templates and Inheritance

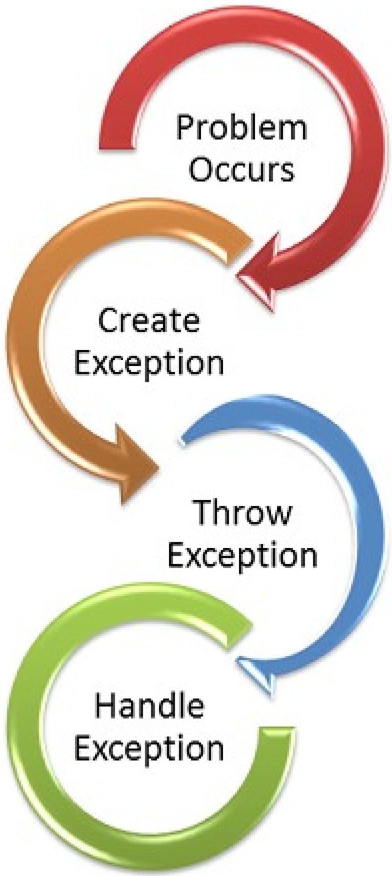
16.5 Introduction to the Standard Template Library

Exception Handling

The best outcome we can ever hope for is when nothing unusual happens.

- However, we need to deal with exceptional things in computer science, and the more of them we handle, the better the user's experience will be.
  - C++ exception handling facilities are used when the invocation of a function may cause something exceptional to occur.
  - Often the exception is some type of error condition.

Exception Handling
4 Steps



Introduction to Exception Handling

Both C++'s library software and code written by outside programmers provide mechanisms which signals when something unusual happens

- This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling* or *catching the exception*

`try-throw-catch` Mechanism

The basic way of handling exceptions in C++ consists of the `try-throw-catch` trio

- The `try` block contains the code for a basic algorithm
  - It tells C++ what to do when everything goes smoothly, no problems or issues

`try-throw-catch` Mechanism

It is called a **`try`** <u>block</u> because it "tries" to execute the case where all goes as planned.

- It can also contain code that *throws an exception* if something unusual happens:

```
try {
        < Code That May Throw An Exception >
}
```

block

`try-throw-catch` Mechanism

Examples of code that can throw exceptions:

- “bad-alloc” when the `new` operator can’t find enough available contiguous memory in the heap
  - creating a string that is too big
- “out-of-range” exceptions when an argument is passed to a function that is too small or large (e.g., an element number in a vector)
- “casting” exceptions when dynamically casting one type of object to another.

`try-throw-catch` Mechanism

- "bad_function_call" when an invalid or empty function is called
- "logic_error" related to the internal logic of a program
- "bad_typeid" when a program receives an identifier for the type of an object which is invalid
- "runtime_error" a catch-all group for all other types of exceptions.

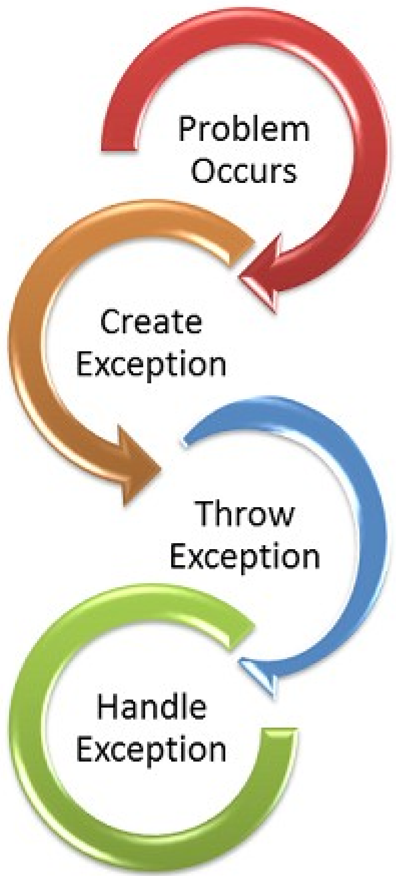
Throwing an `Exception` Explicitly

In addition, an exception can be thrown explicitly by using the **`throw`** statement:

```
throw xxx ;
```

- The data type of ***`xxx`*** determines how the exception will be handled (selected the specific exception handler).

The `try` block



Entering the `catch` Block

When an exception is thrown, the `catch` block begins

- The `catch` block has one parameter
- The *type* of the parameter determines which `catch` block will handle the Exception and is plugged into the `catch` block parameter
- This is called *catching* or *handling the exception*
  - Whenever an exception is thrown, it must ultimately be handled (or caught) by a `catch` block

`try-throw-catch` Mechanism

A `catch` block looks like a function definition that has a parameter of data type thrown

```
catch(int value) {
    < exception handling code >
}
```

It is not really a function definition though, as `catch` is a C++ keyword.

Exceptions – Key Words

- `throw` – followed by an argument, is used to signal an exception
- `try` – followed by a block `{ }`, is used to invoke code that (might) throw an exception
- `catch` – followed by a block `{ }`, is used to process exceptions thrown in a preceding `try` block. It takes a parameter that matches the type of exception thrown.

Throwing an Exception

Code that detects the exception must pass information to the exception handler. This is done using a `throw` statement:

```
throw string("Emergency!") ;
throw 12 ;
```

- In C++, information thrown by the `throw` statement may be a value of *any* type.

Catching an Exception

Block of code that handles the exception is said to *catch* the exception and is called an *exception handler.*

- An exception handler is written to catch exceptions of a given type: For example,

```
catch(string str) {
    cout << str ;
}
```

can only catch exceptions that are string objects.

Catching an Exception

Another example of a handler:

```
catch(int x) {
    cerr << "Error: " << x ;
}
```

This can catch exceptions of type `int`.

Connecting to the Handler

Every `catch` block is attached to a `try` block and is responsible for handling exceptions thrown from that block

```
try {
    // code that may throw an exception
   // goes here
}
catch(char e1) {
   // This code handles exceptions
   // of type char that are thrown
   // in this block
}
```

Exception Example

An example of exception handling is code that computes the square root of a number.

- It will throw an exception in the form of a string object if the user enters a negative number.
- Otherwise, it will calculate and display the square root.

Example

```
int main( ) {
    try {
        double x ;
        cout << "Enter a number: " ;
        cin >> x ;
        if (x < 0)
            throw string("Bad argument!") ;
        cout << "Sqr root of " << x << " : " << sqrt(x) ;
    }
    catch(string str) {
         cout << str ;
    }
    return 0 ;
}
```

Flow of Control

1. The computer encounters a `throw` statement in a `try` block.
2. The computer evaluates the `throw` expression, and immediately exits the `try` block.
3. The computer selects an attached `catch` block that matches the type of the thrown value, places the thrown value in the catch block's formal parameter, and executes the catch block.

Uncaught Exceptions

An exception may be uncaught if:

- there is no `catch` block with a data type that matches the exception that was thrown, or
- it was not thrown from within a `try` block

The program will terminate in either case.

Handling Multiple Exceptions

Multiple catch blocks can be attached to the same block of code. The catch blocks should handle exceptions of different types:

```
try{...}
catch(int int_ex){...}
catch(string str_ex){...}
catch(double d_ex){...}
```

Exception When Calling `new`

If `new` cannot allocate memory, it throws an exception of type (class) `bad_alloc`

- Must `#include <new>` to use `bad_alloc`
- You can invoke `new` from within a `try` block, then use a `catch` block to detect that memory was not allocated.

(Bad Alloc Demo)

Where to Find an Exception Handler?

The compiler looks for a suitable handler attached to an enclosing `try` block in the same function.

- If there is no matching handler in the function, it terminates execution of the function, and continues the search for a handler starting at the point of the call in the calling function.

Unwinding the Stack

An unhandled exception propagates backwards into the calling function and appears to be thrown at the point of the call.

The computer will keep terminating function calls and tracing backwards along the call chain until it finds an enclosing **`try`** block with a matching handler, or until the exception propagates out of **`main`** (terminating the program).

This process is called *unwinding* the call stack.

Nested Exception Handling

`try` blocks can be nested in other `try` blocks and even in `catch` blocks.

```
try {
      try{ … }
      catch(int i){ … }
}
catch(string s)
{ … }
```

Rethrowing an Exception

Sometimes an exception handler may need to do some tasks, then pass the exception to a handler in the calling environment.

- The statement

```
throw ;
```

  with no parameters can be used within a `catch` block to pass the exception to a handler in the outer block.

Throwing an Exception Class

An exception *class* can be defined and thrown when an error condition is encountered

- A catch block must be designed to catch the object of the exception class
- The exception class object can contain information which is passed to the exception handler via its data members

Creating Exception Classes

Create “NegativeWidth” and “NegativeLength” exception classes (inner to Rectangle) thrown and unwound to the exception handlers in **`main`**.

(See instructions in Canvas)

Function Templates

Function template: A pattern for creating definitions of functions that differ only in the type of data they manipulate. It is called a *generic* function.

They are preferred over overloaded functions when the code defining the algorithm (body) of the function can be written only once.

Example

Two functions that differ only in the type of the data they manipulate

```
void swap(int &x, int &y){
    int temp = x ;
    x = y ;
    y = temp ;
}

void swap(char &x, char &y) {
    char temp = x ;
    x = y ;
    y = temp ;
}
```

Differences

A `swap` Template

The logic of both functions can be captured with one template function definition:

```
template<class T>
void swap(T &x, T &y) {
    T temp = x ;
    x = y ;
    y = temp ;
}
```

Using a Template Function

When a function defined by a template is called, the *compiler* creates the actual definition from the template by inferring the type of the type parameters from the arguments in the call:

```
int i = 1, j = 2 ;
swap(i, j) ;
```

- This code makes the compiler instantiate the template with type **`int`** in place of the type parameter **`T`**.

Function Template Notes

A function template is a *pattern.*

- No actual code is generated until the function named in the template is made at compile time.
- A function template uses no memory.
- When passing a *class* object to a function template, ensure that all operators referred to in the template are defined or overloaded in the class definition.

Function Template Notes

- All data types specified in template prefix must be used in template definition.
- Function calls must pass parameters for all data types specified in the template prefix.
- Function templates can be overloaded – need different parameter lists.
- Like regular functions, function templates must be defined before being called.

Where to Start When Defining Templates

Templates are often appropriate for multiple functions that perform the same task with different parameter data types:

- Develop function using usual data types first, then convert to a template:
  - add the template prefix
  - convert data type names in the function to type parameters (*i.e.*, **`T`** types) in the template

Function templates and C++ 11

Starting in C++ 11, the key word `typename` can be used instead of `class` in the template prefix.

- Therefore,

```
template<class T>
```

  may be written as:

```
template<typename T>
```

The `swapMe` Template

Test the `swapMe` template using ints, chars, and strings

(Swap Me)

Class Templates

It is possible to define templates for classes. Such classes define abstract data types

- Unlike functions, a class template is instantiated by supplying the type name (**`int`**, **`float`**, **`string`**, etc.) at *object* definition.

Class Template

Consider the following classes

1. Class used to join two integers by adding them:

```
class Joiner {
    public:
        int combine(int x, int y)
        { return x + y ; }
} ;
```

2. Class used to join two strings by concatenating them:

```
class Joiner {
    public:
        string combine(string x, string y)
        { return x + y ; }
} ;
```

Example Class Template

A single class template can capture the logic of both classes: it is written with a template prefix that specifies the data type parameters:

```
template <class T>
class Joiner {
public:
    T combine(T x, T y)
        { return x + y ; }
} ;
```

No <T> here

Using Class Templates

To create an object of a class defined by a template, specify the actual parameters for the formal data types

```
Joiner<double> jd ;
Joiner<string> sd ;
cout << jd.combine(3.0, 5.0) ;

cout << sd.combine("Hi ", "Ho") ;
```

Template type goes here

Prints **8.0** and **Hi Ho**

Member Function Definitions Outside of a Template Class

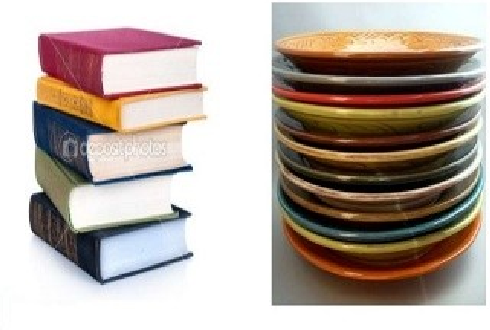
If a member function is defined outside of the class, then the definition requires the template header to be prefixed to it, and the template name and type parameter list to be used to refer to the name of the class:

```
template<class T>
T Joiner<T>::combine(T x, T y)
     { return x + y ; }
```

Class Templates and Inheritance

- Templates can be combined with inheritance.
- You can derive:
  - Non template classes from a template class: instantiate the base class template and then inherit from it
  - Template class from a template class
- Other combinations are possible.

EXAMPLES OF STACK:



Stacks

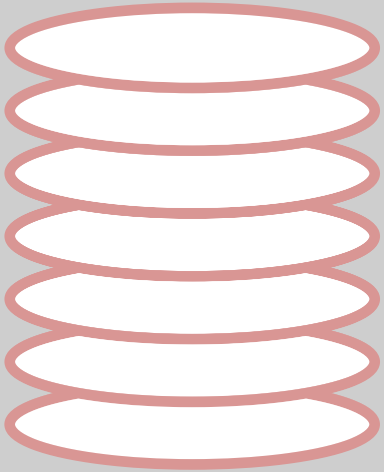
Computer systems often use two types of structures called *stacks* and *queues*.

The most common analogy for a stack is a stack of plates on a spring at a buffet line:

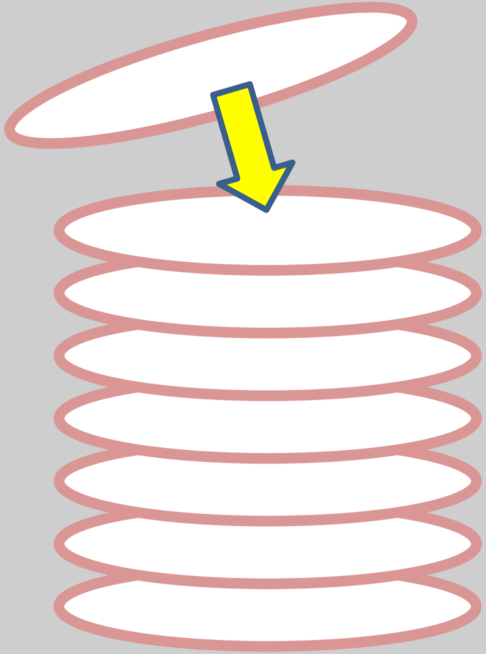
- As plates are washed and dried, they are placed on the *top* of the stack.
- When hungry customers takes a clean plate, they take plates from the *top* of the stack.

Stack Example

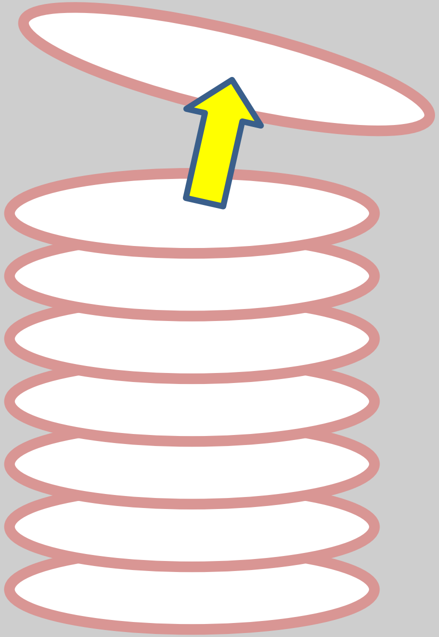
Stack of Plates



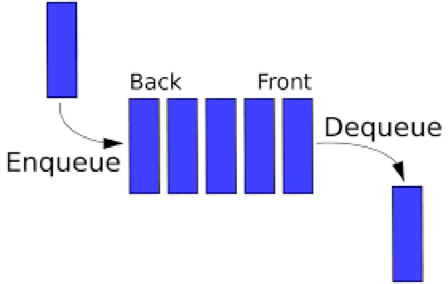
Clean Plates
Goes on *Top*



Hungry Customers
Take New Plates
From the *Top*



Queues



Queues work from opposite sides. A common example is people waiting in line:

- As people enter the queue, they go to the *back* of the line
- When people are serviced, they are taken from the *front* of the line

In fact, “lines” are called “queues” in the UK

Queue Example

Exit
Here

Enter Here

Stacks and Queues

Since the last (or most recent) plate on a stack is the first one out, stacks are often referred to as “last-in-first-out” structures, or

LIFO

Since the first (or oldest) person in a queue is the first one out, queues are often called “first-in-first-out” structures, or

FIFO

Stacks Used In Programs

Windows Explorer or *Finder* – going forward, then back to the last folder or folders

Browsers – going forward, then back to the next page or pages

C++ programs – invoking, then returning from a function or functions

Introduction to the Standard Template Library

Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms.

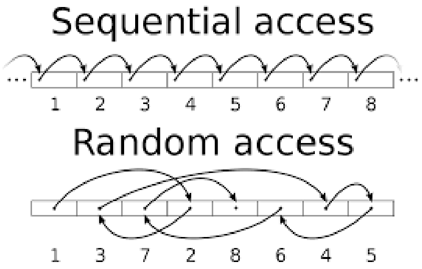
- Programs also can be developed faster and are more portable if they use templates from the STL.

Standard Template Library

Two important types of data structures in the STL:

- *containers*: template classes that store data and impose some organization on it.
- *iterators:* pointers-like objects which provide mechanisms for accessing and traversing elements in a container.

Types of Containers



Two types of container classes in STL:

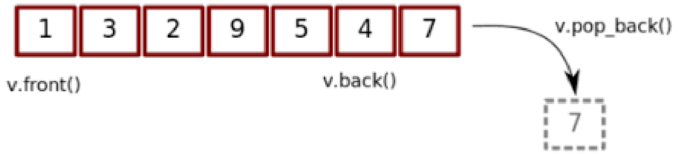
- *Sequential* containers: organize and access data sequentially, as in an array. These include **`vector`**, **`list`**, and **`deque`** containers.
- Sequential container terms: *front* and *back* for locations. If you can access an element in the container without having to go through the other elements, the container provides *random access*.
- These are template classes, so they can store data of any type.

Sequential Containers

Items in sequential containers are typically add to either the “front” or the “back” of the container:

- If you can access an element in the container without having to go through the other elements, the container provides random access. (Not a list)
- To determine if an element is in a sequential container, start at either end and look at each element one-by-one until it is found.

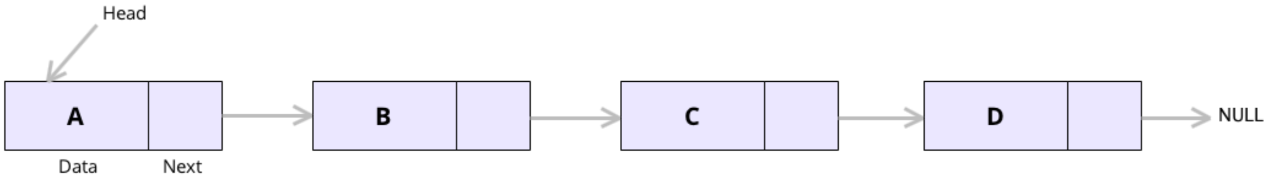
Vectors



Vectors use an underlying array data structure which must grow as items are added or inserted.

- Items are typically pushed onto or removed only from the "back" (or top) of the vector. (Pushing or popping from the “front” is not supported.)
- Adding or inserting into large vectors may require relocating several other elements in the vector.

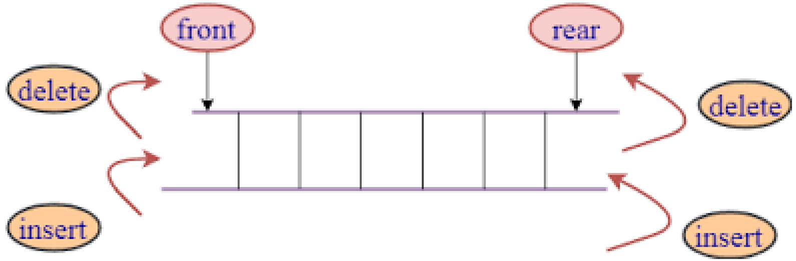
Lists



Lists are stored as individual objects in memory, where each has a reference to the next element.

- Items can be added anywhere in the list by simply manipulating pointers.
  - Adding to the front or back are trivial.
- Lists are slower for searching than vectors since items aren't contiguous in memory, but easier to add or delete items, since other elements do not need to be relocated, only pointers manipulated.

Deques



Deques are usually implemented as a sequence of small arrays which live and die as needed.

- Hence large re-allocations aren’t required.
- A deque can be applied as a combination of a stack and a queue. Items can be added or removed from either end (push_back and push_front, pop_back and pop_front).

Creating Container Objects

- To create a list of `int` elements, write

```
list<int> mylist ;
```

- To create a vector of `string` objects, write

```
vector<string> myvector ;
```

  - Requires the `vector` or `list` header file

Using Sequential Containers

Create and manipulate objects of the vector

and list sequential containers

(Sequential Containers)

Using Deques

Create and manipulate a deque.

(Deque)

Associative Containers

Associative containers use *keys* to access data elements efficiently.

- These containers include **`set`**, **`multiset`**, **`map`**, and **`multimap`**.
- All associative containers store their keys in sorted order for quick look-up (binary searches).
- Like sequential containers, these are templates, so they can store any type of data elements.

Sets

Sets only have keys, no associated data elements.

- In a `set`, keys are *unique*. Keys are not unique in a `multiset`.
- The value of a key in a set cannot be changed, only inserted or removed.
- The order is determined by a `compare` function, which may be implied, such as with numbers or strings, or a template.

Why Are Sets “Associative”

“Associative” usually implies an association of one type of element with another, such as a part number to its description, or a bank account number to the customer and account data.

- Sets only have keys – so what’s the association?
- In this context it refers to how elements are searched:
  - Since sets are ordered “sorted”, a binary search algorithm can be used to find the element.

Use Sets and Multisets

Insert, erase, then re-insert items into a set and into a multiset.

(Sets)

Maps

Types of maps are **`map`** and **`multimap`**.

- Unlike sets, a map has keys and associated data elements. The keys are unique in a map.
  - Keys can have multiple values in a multimap.
- An example of a map is using a part number, bar code, or SKU to access information about an item, or a bank account number to access information about the account and customer.

Stacks and Queues are "Adapters"

Both stacks and queues are implemented by using *another* type of container as their underlying structure:

- The adapter then hides the functionality of the underlying structure but introduces new or renamed functions and features.
- For example, stacks usually limit pushing and popping from a single end of a deque.
- Queues limit pushing to one end and popping from the other.

Iterators

Generalization of pointers. They are used to access information in containers:

- There are many types:
  - forward (uses **`++`**) and reverse (uses ++)
  - bidirectional (uses **`++`** and **`--`**)
  - random-access
  - input (can be used with **`istreams`**)
  - output (can be used with **`ostreams`**)

Containers and Iterators

- Each container class defines an iterator type, used to access its contents
- The type of an iterator is determined by the type of the container:

```
list<int>::iterator x ;
list<string>::iterator y ;
```

`x` is an iterator for a container of type `list<int>`

Containers and Iterators

Each container class defines functions that return iterators:

`begin()` : points to the item at the start of the container

`end()` : points to the location just past the end of the container

```
vector<int> intVect ;
vector<int>::iterator vInt =
                    intVect.begin() ;
```

Containers and Iterators

Iterators support pointer-like operations. If **`iter`** is an iterator, then

- **`*iter`** is the item it points to: this *dereferences* the iterator
- **`iter++`** advances to the next item in the container
- **`iter--`** backs up in the container

- The **`end()`** iterator points *past the end*: it should never be dereferenced

Traversing a Container

Given a vector:

```
vector<int> v;
for (int k = 1 ; k <= 5 ; k++)
    v.push_back(k * k) ;
```

Traverse it using iterators:

```
vector<int>::iterator iter = v.begin() ;
while (iter != v.end()) {
    cout << *iter++ << " " ;
}
```

and prints `1 4 9 16 25`

Iterator Demonstration

Using a vector if **`int`** elements show:

- forward, reverse, and bidirectional
- indexing and offsets (random access)
- assigning values

(Iterators)

Some `vector` Class Member Functions

Function	Description
`front(), back()`	Returns a reference to the first, last element in a vector
`size()`	Returns the number of elements in a vector
`capacity()`	Returns the number of elements that a vector can hold
`clear()`	Removes all elements from a vector
`push_back(value)`	Adds element containing value as the last element in the vector
`pop_back()`	Removes the last element from the vector
`insert(iter, value)`	Inserts new element containing value just before element pointed at by iter

Algorithms

C++ *algorithms* are a collection of functions designed to use ranges of elements through iterators or pointers on STL containers.

- They operate directly on the values in a container, and do not affect the structure (size, allocation, etc.) of the container.
- They <u>can</u> affect the values in the container.
- Include <algorithm> in your program

Types of Algorithms

Non-modifying sequence

for_each, *find*, count, equal, *distance*….

Modifying sequence

copy, copy_n, copy_if, move, swap, replace, reverse, shuffle, *random_shuffle*

Sorting

sort, stable_sort, partial_sort, is_sorted

Binary searches (operates on sorted ranges)

lower_bound, upper_bound, *binary_search*

Algorithms

Merge

merge, inplace_merge, set_union, set_difference,

Heap

push_heap, pop_heap, make_heap, sort_heap

A *heap* is a way to organize the elements of a range that allows for fast retrieval of the element with the highest value at any moment (with pop_heap), even repeatedly, while allowing for fast insertion of new elements (with push_heap).

Algorithms

Partitions

is_partitioned, partition_copy, partition_point

Min/max

min, max, minmax, min_element, max_element

Other

lexicographical_compare

Sorting and comparison functions use `operator<`

Using STL Algorithms

Many STL algorithms manipulate portions of STL containers specified by a begin and end iterator.

- **`distance(iter1, iter2)`** returns the number of elements in the container between **`iter1`** and **`iter2`** (not including **`iter2`**)
- **`find(iter1, iter2, value)`** returns a pointer to the element **`value`** (if present) between **`iter1`** and **`iter2`**

Using STL Algorithms

- **`max_element(iter1, iter2)`** finds max element in the portion of a container delimited by **`iter1`**, **`iter2`**
- **`min_element(iter1, iter2)`** is similar to above

Container / Algorithm Interaction

Create a vector of random numbers, determine its size, and find an element in the vector

(Container Algorithm Interaction)

More STL Algorithms

- **`random_shuffle(iter1, iter2)`** randomly reorders the portion of the container in the given range (has been deprecated in C++14, replaced by **`shuffle`** algorithm)
- **`sort(iter1, iter2)`** sorts the portion of the container specified by the given range into ascending order

`random_shuffle` Example

The following example:

1. stores the squares 1, 4, 9, 16, 25 in a vector
2. shuffles the vector
3. prints its contents

`random_shuffle` Example

Create a vector of sorted numbers,

shuffle, resort and print

(Shuffle And Sort)

Binary Searches

```
bool binary_search(first, last, value)
```

Searches between two iterators in a container **`first`** (inclusive) and **`last`** (exclusive).

- Returns true if any element exists in the range [**`first`**, **`last`**] is equivalent to **`value`**, and false otherwise.

Binary Searches

The elements in the range must then be sorted according to **`operator<`** .

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is especially efficient for random-access iterators.

`binary_search` Example

Create a vector of 100,000 sorted numbers, then perform binary searches to find an element in the vector and an element not in the vector

(Binary Search)

Assignments for Module 13

- Read chapter 16 of textbook (Exception handling).
- Complete current lab and homework assignment.
- Complete any old homework or labs.
- Wrk on final programming or research project.
- Quiz next week up through this Module.