**Review materials are here:** goo.gl/Zz2JG2 (midterm material pages 1-9) and goo.gl/x06BqE (final material pages 1-18)

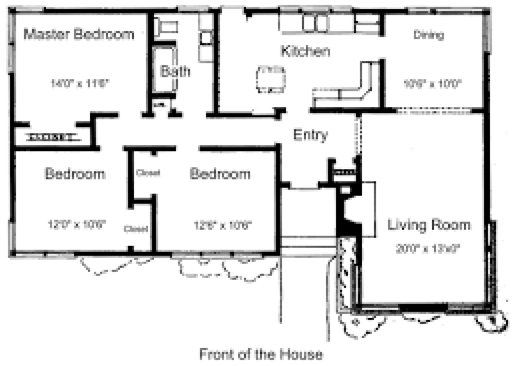
# Chapter 1 - Basics

## Advantages and disadvantages of C++

| Advantages: | vs | Disadvantages: |
|---|---|---|
| --Object Oriented and Procedural | | --Steep learning curve |
| --Programs can compute very quickly! | | --It is not portable |
| --More user control | | --Requires more programmer work |
| --Powerful STL (Standard Template Libraries) | | --For simple programs, procedural languages are better |
| --Code reuse | | --Pointers and memory management →(you need to worry about Garbage Collection) |
| --Easily Maintained | | --Compiler allows the programmer to make mistakes! |
| --Lots of Jobs using C++ | | --C++ can be dangerous → no security/protection against overrunning the buffer and overwriting memory in adjacent locations |
| --Widely supported | | --Requires a strict top to bottom order |

## What is the difference between a class and an object?

| **Class** | **Object** |
|---|---|
| ➔ Class is the blueprint | → object is the instantiation of the blueprint |
| ➔ Declare and define all variables/functions here | → Access functions through an object |



## Inheritance – use to write **reusable** code

Hierarchies – child classes inherit from the parent class

→A toyota is a type of car, therefore toyota inherits from the car class

→Triangles and rectangles are both polygons, therefore they inherit from the polygon class

## Polymorphism → means having many forms

**C++ allows multiple functions to have the same name:**

"refers to the ability to associate many meanings to one function name"– W. Savitch

**Eclipse Example – Overloading.cpp**

## Encapsulation (or Data Abstraction or Information Hiding)

Keep all member variables private, i.e. they are hidden from the user

- Private data cannot be accessed mistakenly by methods outside of the class
- A user or another class cannot directly change the private data; user must go through a public interface to access private data
- Details of how operations work are not known to the user

"The details of the implementation of a class are hidden from the programmers who uses the class" – W. Savitch

## Input and output

```
#include <iostream>                    //used with cin and cout and fixed
Using namespace std;                   // used to replace std::cin or std::cout
```

**cin >> [user input];**

**cout << [variable or object] << endl;**

**Eclipse Example – Print Decimal Places (in Review.cpp)**

# Chapter 3 and 4 – Functions, Paramenters and Overloading

**Eclipse Example – Make Random Numbers (in Review.cpp)**

## What is a Function Signature?

**Only includes:**

1. Name of Function
2. Number and type of parameters
3. Order of parameters

**What it does NOT include:**

1. Return type
2. Pass by value vs pass by reference
3. Keyword const
4. Default arguments

**Do these functions have the same signature?**

```
 int Divide (int n, float m) ;
double Divide (float n, int m) ;
```

## Function Prototypes and Declarations - Declare these at the TOP

- [Return value] [name of function] (list type of parameters - only list these if needed);

## Function Definitions

- Includes the code for the function – describes what the function is supposed to do

## Function Call

- Tells the function to run; it won't do anything unless called

**Eclipse Example – Functions.cpp**

**Eclipse Example – Switch functions (in Review.cpp)**

## Default arguments in a function

1. Default values for the arguments are listed at the **END** of your parameter list

2. int myFunc(int a, int b=6, int c=12);

    a. Are these valid calls?
        i. myFunc(1,2,3) //Valid
        ii. myFunc(1) //Valid
        iii. myFunc() //Invalid!

**Eclipse Example – Default arguments (in Review.cpp)**

## Passing an array as a function parameter

```
void arrayFunc(int arg[]);              //function prototype

int myArray[40];                        //declaring the array

arrayFunc(myArray);                     //calling the function with an array argument
```

**Eclipse Example – Passing arrays and pointers to functions (in Review.cpp)**

## Reference variables

1. Reference variables must be initialized when you declare them
   a. int &refScore ;                    // illegal statement

**int x = 5 ;**
**int z = x;**    $x \rightarrow \boxed{5}\quad z \rightarrow \boxed{5}$

**z = 6;**    $x \rightarrow \boxed{5}\quad z \rightarrow \boxed{6}$

**int &y = x;**    $x \rightarrow \boxed{5} \leftarrow y$    **//y can only be used to refer to an existing integer**

**y = 56**    $x \rightarrow \boxed{56} \leftarrow y$    **//now x=56**

**Eclipse Example – reference variable (in Review.cpp)**

## Enumeration:

The enum is declared as: **enum** enum-type-name enum-variable;

By default, the first enumerator is assigned the integer value 0, and each subsequent enumerator has a value one greater than the previous enumerator:

```
Enum Color {
COLOR_BLACK,                // assigned 0
COLOR_RED,                  // assigned 1
COLOR_BLUE,                 // assigned 2
COLOR_GREEN,                // assigned 3
COLOR_WHITE,                // assigned 4
COLOR_CYAN,                 // assigned 5
COLOR_YELLOW,               // assigned 6
COLOR_MAGENTA }             // assigned 7
```

**Eclipse Example – enumeration (in Review.cpp)**

# Chapter 6 and 7 – Classes and Constructors

**What do Constructors do?**
- Assign data to member variables and any other ‘setup’ that is required such as allocating resources
- Similar to an init function

**Are Constructors defined in a similar way as member functions?**
- Yes, except they must have the same name as the class (Uppercase is convention for class and CTOR)
- Cannot have a return value (not even void)
- Must be public

**When do Constructors get called?**
- When instantiating an object, the object gets ‘constructed’
  - Call to the constrcutor is made every time a new object is created

**Can constructors be overloaded?**
- Yes, similar to function overloading (but again, no return type)
  - Define a default constructor which takes no arguments
  - Define other constructors which take any number of arguments

**Will the default constructor always be called?**
- If (there are no constructors defined }{
  - -> then yes }
- else {
  - at least one constructor has been defined -> then no and you should define a default constructor}

**If there is no default constructor defined, but a constructor taking 1 argument is defined as follows:**

ThisClass {
private string response = “ is the best class”;
public: ThisClass (string semester) { response = semester + response; };
**So, is this a valid call?** -- ThisClass rocks;

**How is the default constructor called?**
- With no parameters as an implicit call or empty paranthesis when using an explicit constructor call

**What happens in the Constructor Initialization list? – See CTORInitialization.cpp**
- Member variables are assigned prior to the constructor body

**What does a destructor do? – Example included with static variables under WorldPeople.cpp**
- “Destroys” an object as it goes out of scope
- Commonly used to de-allocate memory such as when using pointers

**When are destructors called?**
- It will be called automatically when an object is no longer in scope (i.e. program/function ends)
- Or when you call delete (not needed unless working with pointers)

**What is a copy constructor?**
- Creates an object by initializing it with an object of the same class (other object created previously)
  - Used to initialize one object from another of the same type
  - Or copy an object to pass it as an argument to a function
  - Or copy an object to return it from a function

**What is the difference between an Explicit vs Implicit Constructor call?**

- Refer to Jackie's slides
  - Implicit call -> examples Box b1; Box ex(1,2,3); -- both of these are implicit
  - Explicit call -> ex = Box(2,5,8);

## When to use the dot . operator ?

After creating an object, you can access any of its member functions by using the dot operator

```
Car audi;
audi.getMileage();                                  // now you can use the dot operator
```

## When to use the scope resolution operator :: ?

1. Used in place of *using namespace std* – i.e. std::cout
   a. float f = 5.6289;
   b. std::cout << std::setprecision(5) << f << std::endl;

2. Used when functions are defined outside of the class
   a. double Box::getVolume(void) { return length*width*height; }

3. Resolve ambiguity between functions with same template which are derived from different classes
   a. See Scope_ResolveAmbiguity.cpp

4. To override the overridden function
   a. See Override_Function.cpp

5. To access static data members
   a. See StaticMemberFunctions.cpp

6. Unary scope resolution operator - Can refer to global variable within a local function
   a. int x = 5;
   b. int main() {

```
        int x = 1;
        cout << "The variable x: " << ::x << endl;   //  prints 5
        cout <<"The variable x: " << x << endl;  // prints 1
        return 0; }
```

**const modifier – used for variables when you don't want them to be changed/modified

**Eclipse example for static variables** – these must be initialized outside of the class

See WorldPeople.cpp and Static Members.cpp for examples of static variables

# Chapter 7 –Vectors

**Vectors** – expandable, dynamic array

1. Good for quick random access
2. Slow to insert & erase in the middle or beginning
3. Quick to insert and erase at end

## Common Vector functions – all public member functions

| | |
|---|---|
| size() : | Return size -- there's also resize(n) which resizes vector to n |
| capacity() : | Returns largest # of items that can be currently stored (is a function of $2^n$) |
| empty(): | Test whether vector is empty – returns boolean value |
| at(index): | Access data at any position |
| assign(): | Assign vector content – can take 1 or 2 arguments depending on version used |
| push_back(value): | Adds element value at the end |
| pop_back(): | Delete last element |
| insert(index,value): | Insert element at specified index and puts in value |
| erase(position): | Erase elements at position specified (or range provided) |
| clear(): | Clears all content; size() = 0 after executed |

**Example – Vectors.cpp and copy_algorithm.cpp**

# Chapter 8 –Operator Overloading and Friends

## Operator Overloading – used to make code easier to read/maintain -- *syntactic sugar*

Operator overloading can be done on everything except for

| | | |
|---|---|---|
| 1. Dot operator | -> | . |
| 2. Scope resolution | -> | :: |
| 3. Pointer to member | -> | .* |
| 4. Ternary | -> | ?: |

The following set of operators is commonly overloaded for user-defined classes:

| | |
|---|---|
| = | (assignment operator) |
| + - * | (binary arithmetic operators) |
| += -= *= | (compound assignment operators) |
| == != | (comparison operators) |
| << >> | (stream operators) |

See Jackie's powerpoint slides on **Operator Overloading and Friends.** Make sure you understand:

1. When and how to use **friends** – (when operator overloading is a non-class member function)
    a. **friends** is *only* in function declaration or inline functions (not repeated again outside of class)
    b. If declared as a friend function, then you must have 2 arguments listed
        i. If not a a friend function, then it is assumed the **this* is on the lhs and the object you want to do something to is on the rhs (See short syntax example below)
2. Why and where to use **const** – protect parameters and return type from being modified
    a. Use when you don't want an object to be accidentally modified/overwritten
3. Must use keyword **operator** and then symbol(s) that are beng overloaded
    a. ALWAYS used at beginning of function definition (after listing the class type)
4. Why overload an operator?
    a. Because objects are user-defined data types and operators will only work on primitive or built-in data types

**Syntax for overloading += operator**

```
// if declared as an inline friend function
// lhs = left-hand side and rhs = right-hand side
// const was used to protect MyClass objects from being modified

friend MyClass operator+=( const MyClass& lhs, const MyClass& rhs ) {
        MyClass result;
        // code here
        return result; }

// same inline function as above, but not a friend function this time
MyClass operator+=( const MyClass& rhs ) const {
        MyClass result;
        // code here
        return result; }
```

**Syntax for overloading << (insertion operator) and >> (extraction operator)**

```
ostream & operator << (ostream &out, const SomeClass &someExample) {
      statements;
      return out;
}
istream & operator >> (istream &in, SomeClass &someExample) {
      statements;
      return in;
}
```

**Example – OperatorOverloading.cpp**

# Chapter 9 –Strings

## Advantages of C++ vs C strings

- C++ strings are mutable! (mutable means modifiable/can be changed)
- Copying, comparing, and assigning using = is much easier in C++
- Default constructor initializes a string object to an empty string **and** uninitialized strings are okay
- Being out of bounds will not cause a problem, but you will access garbage
  - Do not end with the null terminator '\0' like they do with C Strings

**Syntax**
#include <string>
string lastName;                    // constructs an empty string

**3 ways to assign**
lastName = "Davidson";
string lastName("Davidson");
string lastName = "Davidson";          // use without declaring variable above

**String manipulation**
lastName[0] = 'M';                    // lastName = Mavidson

**String concatentation**

```
string firstName(“Alex”);
string lastName(“Perez”);
string fullName = firstName + lastName;          // + operator combines strings objects together
string error = “Michelle” + “!”;                 // cannot combine 2 C strings like this
```

**More string manipulations and comparing**

| | |
|---|---|
| str[i] | Returns the ith character from the string |
| s1 + s2 | Returns a new string of s1 concatenated with s2 |
| s1 = s2; | Copies contents of string s2 into string s1 |
| s1 += s2 | Appends contents of s2 to end of s1 |
| s1 == s2 | Compares two strings lexographically (similar for <, <=, >, >=, !=) |

**String member functions**

| | |
|---|---|
| s1.length() | Returns the number of characters in the s1 |
| s1.at(index) | Returns the character at position index |
| s1.substr(pos, len) | Returns substring of s1 from position pos to length len (not including value at len) |
| s1.find(ch, pos) | Returns first index greater than or equal to pos, containing char ch |
| s1.find(text, pos) | Same as above, except searches for string text instead of character ch |
| s1.insert(pos, text) | Inserts the characters from text before index position pos |
| s1.replace(pos, count, text) | Replaces count characters from text starting at position pos |
| s1.erase(n) | Deletes everything after nth character |
| s1.erase(i,j) | Deletes j number of characters starting at index i |

**How do we get delimited inputs from a user at once?**

cin uses whitespace as a delimiter between inputs, can be a problem when reading string input from user

- Use getline() function
- getline(source, destination) where source is cin and destination is the variable to store string into
- Can grab multiple lines at once by using a delimiter character with getline
    - getline(cin, str, ‘?’) where ? is the delimiter for end of input
    - OR getline(cin, str, ‘,’) for comma separated values

C++ strings should usually be passed by reference (string &s1) and if not modified within the function they should be passed as a const (const string &s1)

**Example – Strings.cpp**

**Comma Separated Values**

- Continuously collect information from the user and separate each input with a comma

# Chapter 14 –Inheritance

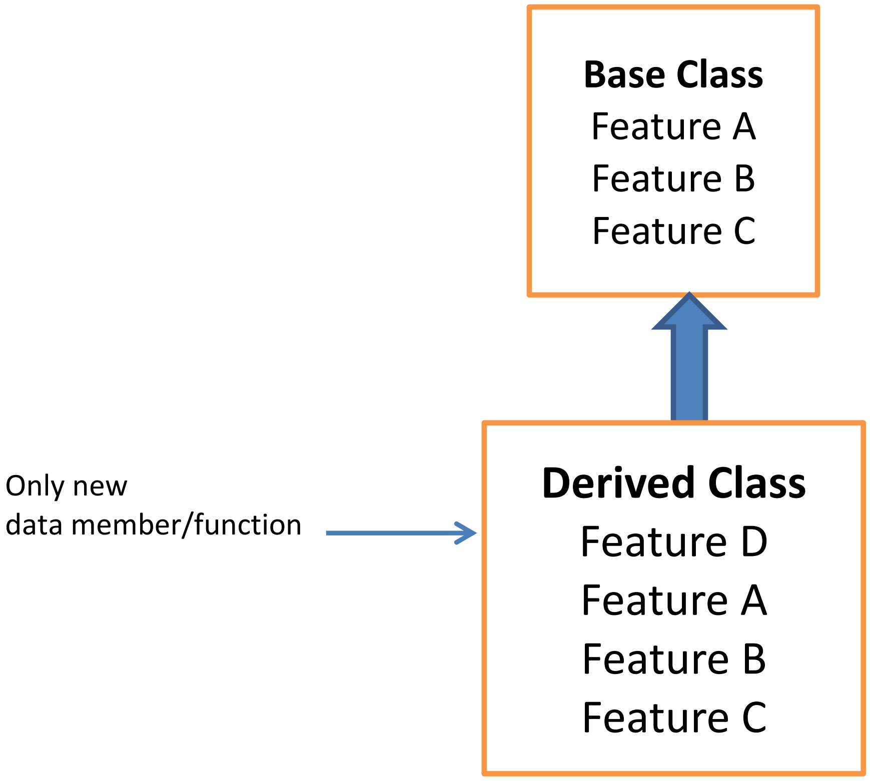
**Inheritance is a Key element of OOP**

- Allows you to derive a new class from an existing one
- The "child" class inherits/has access to data members and member functions of "parent class"

**Advantages**

- Powerful feature of OOP
- **Code reusability !!**
  - Saves you time, money and increases program readability and reliability
- Can distribute class libraries – a programmer can use original class created by another person or company and only modify/change derived classes for particular situations

The arrow represents an -- IS A relationship



After implementing this diagram, what features does the Base class have access to? What features does the Derived class have access to?

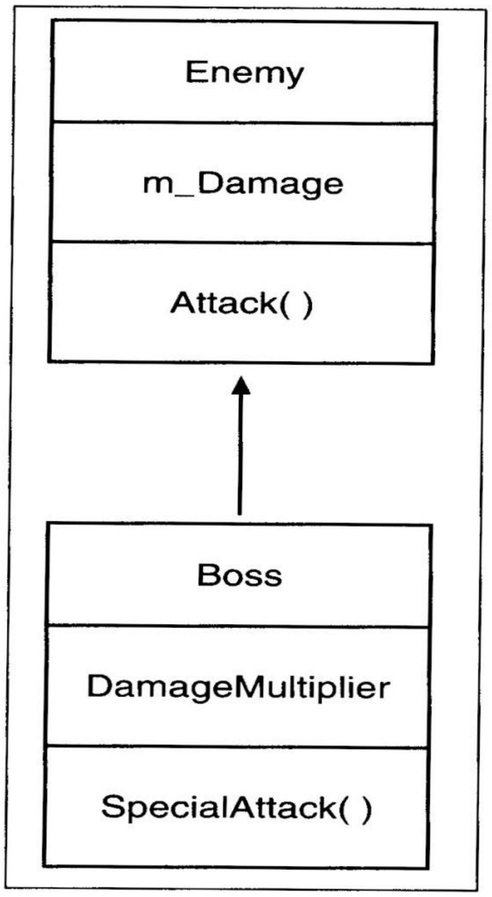
**Example – inherit.cpp**

**What does not get inherited?**

- Constructors
- Destructors
- Copy constructors
- Overloaded assignment operators

**Example – inherit_values.cpp**

**Another inheritance example**



1. **Which ones is the Base class? Which one is the derived class?**
2. **What data features does Enemy have access to?**
3. **What data features does Boss have access to?**

**Inheritance Review:**

Will the derived class inherit the base class constructors?
- No

When you instantiate a derived class, what constructor is called first?
- The base class constructor is called automatically when an object of a derived class is called, so it gets called first and then the derived class constructor

What are some advantages of inheritance?
- See above

What about when using destructors? Which one gets called first?
- Call derived class destructors, then base class destructors – it is the opposite rule to constructors

**Function Overriding vs Function Overloading – (both of these are examples of polymorphism)**

- Function Overloading – creating multiple versions of a function with different signatures
- Function Overriding – provide a new definition of the function in a derived class

Why use Function Overriding?

- In the Enemy and Boss classes, Enemy has Attack() function, but what if you want the derived class to have a special kind of Attack() function that is different than enemy?
    - Can call on base class variables and access/modify them from within the derived class

**Example – inherit_overloading.cpp**

# Chapter 15 – Polymorphism and Virtual Functions

**Virtual functions** – (also called late binding as it occurs during runtime)

- Allows programmer to declare a function, but not define it within the base/parent class
- Derived classes can define virtual function during runtime when there is an instance of that object

Examples of polymorphic binding

- Double-clicking with a mouse
    - Double-click on a file folder - > result is that a directory opens
    - Double-click on web browser -> result is a web browser is launched
    - Double-click on a word icon -> word processing is launched
- 3 separate double-clicks produced 3 different results – polymorphic run time binding

**Example – virtual.cpp**

**Abstract base classes**

- Must have at least one pure virtual function
- A pure virtual function has =0 at the end and it does not have a definition
- Generic classes which will not be instantiated

**Static binding vs Dynamic binding**

- Static - > code for the function is attached at compile time
- Dynamic - > code for the function is attached at runtime (happens with virtual functions)

**Example – abstract.cpp**

# Chapter 19 – Standard Template Library

**The Standard Template Library**

**Definition:** The Standard Template Library, or *STL*, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template.

**Advantages/Facts**

- Allows a function or class to work on many different data types without being rewritten for each one.
- Saves time not having to retype for all possible types (e.g. one for ints, one for doubles, one for floats)
- What is the advantage of using the STL?
    - Saves time not having to reinvent (or retype) the wheel. Elegant, Efficient, Versatile.

**Example – template_example.cpp**

## Definition of Container, algorithm, and iterator

**Container:** A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allow a great flexibility in the types supported as elements. The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers). Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

**Algorithm:** collection of functions especially designed to be used on ranges of elements.

**Iterator:** The concept of an iterator is fundamental to understanding the C++ Standard Template Library (STL) because iterators provide a means for accessing data stored in container classes such a vector, map, list, etc.

```
Iterator example
      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
            cout << *it << endl;

      set<int>::iterator it // Declares the iterator
      *it // Dereferences the iterator, to use the value there.
```

**Sets**

- Sets are an associative container that stores unique elements (the elements are the keys)
- Example: set<int> numbers;

**Advantages/Facts**

- A big feature of using a Set is because it “gets rid of duplicates” – You can not insert something into a set if it already exists in the set.
- A set is like a map without values – they key is the value in this case
- A set is concerned whether an element is contained in the set.
- Maps and Sets don’t support operations such as push_back() and pop_back(). The reason is that for both sets and maps, the set and map determine where a new value is entered, not the programmer.
- Sets are stored in balanced binary trees. – Advantage of binary search versus linear search is great.
- A set node contains the key first as well as a pointer to the left and right nodes.
- Multisets allow multiple keys.

**High Level Operations: set_union, set_intersection, and subset**
#include <algorithm>

The ***union*** will build a set that contains all the elements that occur in at least one of the sets.

```
OutputIterator set_union (InputIterator first.begin(), InputIterator first.end(),
                          InputIterator second.begin(), InputIterator second.end(),
                          OutputIterator result);
```

The ***intersection*** of two sets is formed only by the elements that are present in both sets.

```
OutputIterator set_intersection (InputIterator f.begin(), InputIterator f.end(),
                                 InputIterator s.begin(), InputIterator s.end(),
                                 OutputIterator result);
```

The ***difference*** of two sets is formed by the elements that are present in first set, but not in the second one.

```
OutputIterator set_difference (InputIterator f.begin(), InputIterator f.end(),
                               InputIterator s.begin(), InputIterator s.end(),
                               OutputIterator result);
```

**Example – set_example.cpp**

<u>Sets are optimized for fast searching of elements, thus they have special search operations.</u>

| | |
|---|---|
| **find**(elem); | //Returns the position of first value of elem or end() |
| **lower_bound**(elem); | //Returns the first position, where elemwould get inserted (the first element>=elem) |
| **upper_bound**(elem); | //Returns the last position, where elemwould get inserted (the first element>elem) |

**Iterators:**

| | |
|---|---|
| begin(); | Return iterator to beginning (public member function ) |
| end(); | Return iterator to end (public member function ) |

**Capacity**:

| | |
|---|---|
| empty(); | Test whether container is empty (public member function ) |
| size(); | Return container size (public member function ) |
| max_size(); | Return maximum size (public member function ) |

**Modifiers**:

| | |
|---|---|
| insert(); | Insert element (public member function ) |
| erase(); | Erase elements (public member function ) |
| swap(); | Swap content (public member function ) |
| clear(); | Clear content (public member function ) |

<u>**Maps:**</u>
Maps are an associative container that stores unique keys that are paired with (not necessarily unique) values.
Think <Key, Value> pairs.
i.e.
<Person, Age> or
<Word, Number of occurrences> or
<Day, Temperature>
Maps map <Some Key,> to <, Some Value>

**Facts:**

- There cannot be duplicate keys. If you reassign the value at a key, it will overwrite.
- STL implementation is a balanced binary search tree
- An ordered container of (key, value) pairs -> elements are put in order of key by default

One common way to use maps is as associative arrays.
In an ordinary C++ array you use the array index to access an element.
An associative array works in a similar way except that you can choose the data type of the array index. e.g.
myMap[“Key”] = “Value”;
You can use std::make_pair(key, value); to insert data into a map. E.g. map.insert(make_pair(key,value));

For both map and multimap, the keys are inserted in ascending order.
This can be overridden
map <int, string, **greater<int>** > map;

**Deque:**

- Deque implements a double-ended queue with comparatively fast random access
- You can insert/delete items at the end and beginning efficiently compared to vectors
- Cannot control the capacity of the container and the moment of reallocation of memory
- Supports random access iterators, and insertion and removal of elements invalidates all iterators to the deque

## Difference between a deque and a vector:

| Vector | Deque |
|---|---|
| Elements are stored contiguously.<br>Initial capacity can be denoted.<br>Random access<br>Low memory usage | Uses blocks of memory.<br>Can not control memory reallocation.<br>Access from each end. |

**Characteristics, advantages, and disadvantages of vector, list, and deque.**

**Vector**:

- Relocating, **expandable dynamic array**.
- Always occupies a contiguous region of memory.
- Do not need to know the size at compile time
- **Quick random access** by index number. [i]
- **Slow to insert and erase in the middle.**
- **Quick to insert and erase at the end.**

**List**:

- A double linked list, each element contains a pointer to the next element and the element in front of it
- **Quick to insert and delete at both ends**.
- **Quick access at both ends**.
- **Slow random access**.
- The container stores the address of both the front and the last elements, which makes for fast access at both ends
- Compared to vectors and deques, **lists are efficient for inserting and deleting elements anywhere in the list**.
- Thus, for sorting algorithms lists are efficient.
- Lists do not have the [] operator and **cannot do random access**.
- In order to access the 1000 element, they would have to iterate through all 1000 elements.

**Deque:**

- Is like a vector but can be accessed at both ends.
- **Quick random access** (using index number)
- **Slow to insert and erase in the middle**
- **Quick insert or erase** (push or pop) **on both ends**
- A deque can expand in both directions, front and back.
- The term deque stands for **doubly ended queue**.
- A deque is stored in segments, in non-contiguous areas.

**What is the difference between a map and a set?**

- Maps store Key Value Pairs.
- Sets store Keys

**What is a set and a multiset. How are elements stored?**

- Sets are containers that store unique elements following a specific order.
- Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.
- Both are typically implemented as *binary search trees*

**For both maps and sets know how to place data in ascending or descending order.**

- Can only be done during the constructor call.

**Ascending is low to high. Descending is high to low. What is the default for map and set?**

- Ascending. Both defaults to less<T>, which returns the same as applying the less-than operator (a<b).

**What is the difference between an associative and sequence container?**
**Sequential containers**:
- The elements can be accessed sequentially
- Are more efficient in accessing elements by their position
- Fast random access
- Where one element follows another element.  You can visualize the elements as a line, like houses on a street.
- The sequence containers include:  vector, list, and deque.

**Associative containers**:
- Designed to be especially efficient in accessing its elements by their key
- Logarithmic time - O(log *n*)
- Are typically implemented using self-balancing binary search trees
- Uses keys to access data.
- The associative containers are sets, multisets, maps, and multimaps.  The associate containers store data in a structure called a tree.

**What is the most useful STL?**
- vector

**What is the second most useful STL?**
- map

**Know how to use the copy algorithm.**
Copies the elements in the range [first,last) into the range beginning at *result*.

**Data Types**

| | | | | |
|---|---|---|---|---|
| **char** 1 byte | **bool** 1 byte | **short** 2 bytes | **int** 4 bytes | **unsigned int** 4 bytes |
| **float** 4 bytes | **long** 8 bytes | **double** 8 bytes | **long double** 16 bytes | |

**Access Control:**

| **Private** | **Protected** | **Public** |
|---|---|---|
| Class members declared as **private** can be used only by member functions and friends (classes or functions) of the class. | Class members declared as **protected** can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class. | Class members declared as **public** can be used by any function. |

## Pointer arithmetic

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
  int  var[MAX] = {10, 100, 200};
  int  *ptr;

  // let us have array address in pointer.
  ptr = var;
  for (int i = 0; i < MAX; i++)
  {
    cout << "Address of var[" << i << "] = ";        Address of var[0] =
    cout << ptr << endl;                             0xbfa088b0

    cout << "Value of var[" << i << "] = ";          Value of var[0]
    cout << *ptr << endl;                            10

    // point to the next location
    ptr++;
  }
  return 0;
}
```

| **Var=ptr** | **0** | **1** | **2** |
|---|---|---|---|
| **address** | **0xbfa088b0** | **0xbfa088b4** | **0xbfa088b8** |
| **value** | **10** | **100** | **200** |

| Value of i | **(Pointer Arithmatic)** Address of array locations presented by **ptr** incremented by 4 | Actual value in the array presented by ***ptr** |
|---|---|---|
| 0 | Address of var[0] = 0xbfa088b0 | After ptr=var  Value of var[0] = *ptr = 10 |
| 1 | Address of var[1] = 0xbfa088b4 | After ptr++  Value of var[1] = *ptr = 100 |
| 2 | Address of var[2] = 0xbfa088b8 | After ptr++  Value of var[2] = *ptr = 200 |

**//when is = 0**
Address of var[0] = 0xbfa088b0
Value of var[0] = 10

**//when is = 1**
Address of var[1] = 0xbfa088b4
Value of var[1] = 100

**//when is = 2**
Address of var[2] = 0xbfa088b8
Value of var[2] = 200

**Review materials are here:** goo.gl/Zz2JG2 and goo.gl/x06BqE