# A brief introduction to STL vector and string

# What is STL?

- STL: Standard Template Library
- It is part of the standard C++ library.
- STL provides the following:
  - Containers
    - Example: vector, list, map, set
  - Iterators
    - For iterating over elements of containers
  - *Algorithms*
  - *Allocators*
  - *Function objects*
- STL is a generic library.

# Containers in STL

- Containers are essentially a collection of objects.
- STL has different types of containers that can be used depending on the requirements. Each has its own advantages and disadvantages.
- Following are some of the container classes available in STL:

1. `vector`
2. `list`
3. `deque`
4. `set`
5. `multiset`
6. `map`
7. `multimap`

# Containers

- These containers fall into two categories
  - Sequence containers:
    - These contain the objects in the order that they are inserted. So, the value of the objects inserted does NOT have any bearing on where these objects are placed in the container.

      `vector, list, deque` are sequence containers
  - Associative containers:
    - These contain the objects NOT in the order that they are inserted. The value of the objects inserted DOES have a bearing on where these objects are placed in the container.
    - These are sorted collections, where sorting happens based on a certain specified criteria (if not specified, there is a default criteria).

      `map, multimap, set and multiset` are associative containers.

# Containers...

- Since associative containers are sorted collections, search of an object is a binary search, and not a linear search as would normally be the case with sequence containers.

- Vector is essentially a dynamic array.
- List is the typical list data structure that you would have read about in a data structures class.
- Map is essentially a balanced binary tree. It contains a (key, object) pair. Sorting (and searching) is done based on the key, not the object.
- Multimap is a map that allows duplicate keys. In a map, there cannot be more than one entry for a given key.
- Set is like a map, except that it does not contain the (key, object) pair. In this, the objects are sorted based on their own values.
- Multiset is a set that allows duplicates (i.e., objects that have the same value).

# Some definitions before we proceed...

- STL containers place certain requirements on the containing types.
- Its good to understand what these requirements mean.

- Assignable
  - A type is assignable if
    - you can assign to objects of that type,
    - make copies of objects of that type.

- Default constructible
  - A type is default constructible if it has a default constructor (i.e., a constructor that can be used without having to specify a value).

# Definitions...

- Equality comparable
  - A type is equality comparable if its objects can be compared using the operator ==. For class types, it means that the operator == must be overloaded for that class.
  - Note that the operator== has to be an equivalence relation.
  - Equivalence relation means that the following are satisfied:
    - Reflexive:
      - ObjectA == ObjectA
    - Symmetry:
      - ObjectA == ObjectB follows from ObjectB == ObjectA
    - Transitive:
      - If ObjectA == ObjectB,
        And
        If ObjectB == ObjectC
        Then
        ObjectA == ObjectC.

# Definitions...

- Less than comparable
  - A type is Less than comparable if it is possible to compare two objects of that type using the operator <.
  - For class objects, the operator < has to be overloaded.
- Copyable
  - A type is copyable if you can create a copy of an object of that type.

# Vector

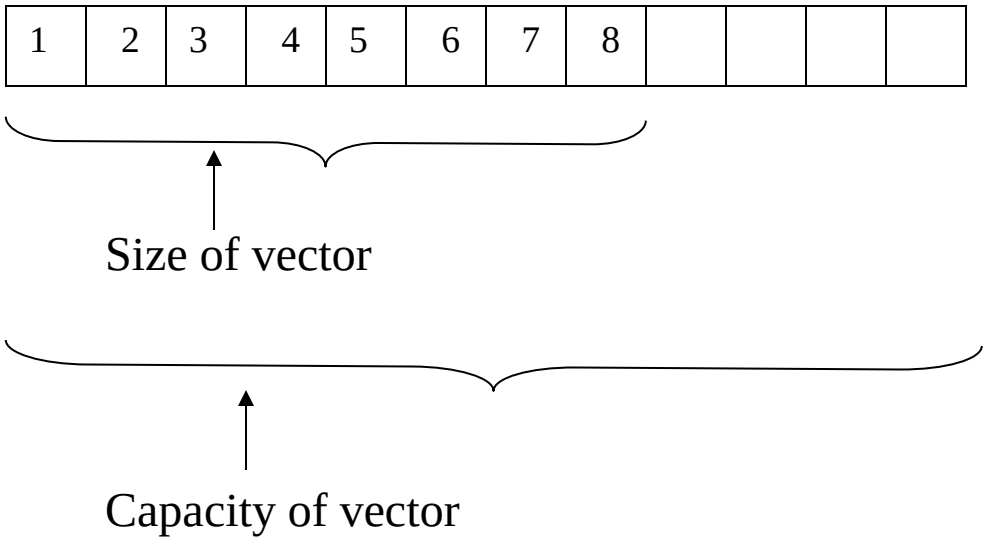
- A vector is a dynamic array.
- It allows random access, so the access complexity is O(1).
- Elements in the vector should be assignable and copyable.

# Vector

- Appending (i.e., inserting at end) and removing elements at the end is fast, since random access is allowed.
- Inserting or deleting within the vector is slow, since elements need to be shifted to the right (left for deletion).
- Insertion or deletion at the first position is the most expensive.

# Vector

- Vector is essentially contiguous memory (although not specified as such in the standard).




- Size is the current number of elements in the vector.
- Capacity is how many it can hold before it needs to allocate more space.
- Capacity is >= size

# Vector

- Member method `reserve` can be used if you want to specify (and know) the size. This ensures that reallocation does not happen while inserting elements.

```
vector< int > vi;
vi.reserve( someSize );
```

- OR

```
vector< int > vi (someSize);
```

Both forms will make the capacity equal to **`someSize`**.

However, in the second case, the default constructor is called for every element created, but not so in the first case.

```
vector< MyString > vi (someSize);
```

This statement will not compile if class MyString … ?

# Vector

- vi.size() returns number of elements in the vector.

- vi.capacity() returns the number of elements the vector can hold WITHOUT having to reallocate memory.

```
vector< int > vi;
vi.reserve( numElements );
```

- OR

```
vector< int > vi (numElements );
```

In the first case, vi.size() will return 0, in the second case, vi.size() will return `numElements` .

In the first and second case, vi.capacity() will return a value >= `numElements`

# Simple example

```
#include <vector>
using std::vector;
int main()
{
    vector <int> vi;        // a vector of integers, or dynamic array of integers

    for (int ii = 0; ii < 200; ++ ii)
         vi.push_back( ii );                  // vi will grow if needed

    int vecSize = vi.size();

    for (int ii = 0; ii < 200; ++ ii)        // ideally ii < vi.size()
         cout << "Integer at position " << ii << " is " << vi[ii] << endl;

    return 0;
}
```

# Swapping vectors

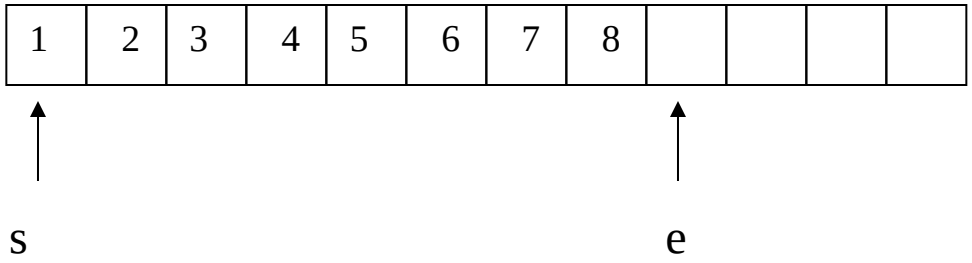
Example showing how to swap two vectors:

```
vector< MyString > v1;
vector< MyString > v2;

//... Code to add elements to v1 and v2

std::swap(v1, v2 );
//OR
v1.swap( v2 );
```

- swap(v1, v2) or v1.swap( v2 ) will swap the two vectors in constant time, i.e., the swap complexity does NOT depend on the sizes of the two vectors.
- This constant complexity of swap method is true in general for all other swap methods in STL (e.g. string, set, etc.)

# Half open range

- A half-open range is a range that includes all values starting from (and including) the first value, up to (but not including) the last value.




- *s* and *e* are pointers to the elements in the array.
- *s* points to the first element. *e* points to one past the last element.
- *s* can be dereferenced, but its an error to dereference *e*.
- However, *e* can be used for comparisons against other pointers in this container. It essentially acts as a sentinel value, signifying the end.

- This is a half open range (sometimes called semi open range).
  - Its written as [s, e)
  - Its very commonly used in STL for iterating over containers.

# The string class in C++

- C++ Standard Library provides a string class that encapsulates all of the functionality that you might have seen in C string library functions, and more.
- The standard specifies a standard public interface, but not the implementation.
- As a result, a string implementation may or may not have reference counting, which can improve performance.

# Constructors

- string s1; // empty string
- string s2 = ""; // empty string
- string s3 ("Test"); // conversion from character literal
- string s4 ( s3); // copy construction
- string s5 = 'a'; // ERROR ! No such conversion
- string s6 = 10; // ERROR ! No such conversion
- string s7 ( 10, 'B'); // creates "BBBBBBBBBB"
- string s8 ( s3, 1, 2); // creates "es"

- string s9 ( s7, 2, 5 ); // creates "BBBBB"
- string s9 ( s7, 2, 50); // from position 2 to end of s7
- string s10 ( s7, 2, string::npos); // from position 2 to end of s7
- string s11 ( s7, 50, 2); // throws out_of_range()

# Length of string

- length() and size() return the number of characters in the string.
- s7.length()  returns  10
- Length of all strings is  less than  string::npos.
- If you try to construct a string that is too long, then a length_error is thrown.

# Accessing characters

- The [ ] operator can be used to access an element (character) from a string.
- This operator does not do any range checking. If index is out of bounds, behavior is undefined.
- at() member can also be used to access elements from a string. It does range checking, and throws out_of_range() if index is out of bounds.
- at() is slower than operator [ ]. So, if you are sure index is within bounds, use [ ], otherwise use at() operator.

# Accessing characters

- Example:
  - string s1 ( "Access");
  - char ch1 = s1 [ 0 ]; // ch1 has 'A'
  - char ch2 = s1 [ 5 ]; // ch2 has 's'
  - char ch3 = s3 [ 10 ]; // undefined behavior
  - char ch4 = s3.at( 0 ); // ch4 has 'A'
  - char ch4 = s3.at( 6 ); // throws out_of_range()
    - NOTE : 6 == s1.length()
  - char ch4 = s3.at(10); // throws out_of_range()

# Accessing the C type string

- A string object lets you access the C type NULL terminated string.
- This can be used for interfacing to older functions (or C functions) that expect a NULL terminated string.

- ```
  string s1 ("Access");
  const char * strPtr = s1.c_str();        // returns NULL terminated string
  ```

- The pointer returned by c_str() is owned by the string object.
- This value should NOT be modified or deleted by the caller.
- This value is valid only until
  - the string object s1 exists.
  - Only const member functions are called on it.
  - No new value is assigned to it. Ex. s1 = string ("Another");
  ```

# Example

```
const char * s1Ptr ;
const char * s2Ptr;

if (some condition)
{
      string s1 ("Access");

      s1Ptr = s1.c_str();              // s1Ptr points to NULL  terminated "Access"
      s1 = string ("Another");         // s1Ptr is not valid after this statement.
      s2Ptr = s1.c_str();              // s2Ptr points to NULL  terminated "Another"
}
```

s2Ptr is **not** valid outside the block, since s1 does not exist.

Either s1Ptr and s2Ptr should be declared inside the block (same lifetime as string s1,

OR

they should be set to NULL before exiting the block ( NOT deleted, just set to NULL).

# Appending

- Appending to a string can be done in one of two ways :
  - operator += (arg)
  - append ( arg )
    - where arg can be any one of following :
      - const string & <-- another string
      - const char * <-- pointer to character string
      - char <-- character. This CANNOT be used with append. Instead the option is string::push_back ( char c )
  - These members throw length_error() if the resulting string will be of length greater than string::npos
  - Note : There are more signatures of append that take number of characters to append and starting index.

# Erase and Clear

- void string::clear()
- string & string::erase()
  - Both of these can be used to empty the string.
  - erase() returns a reference to self.

- string & string:: erase( size_type  startingIndex )
- string & string:: erase( size_type  startingIndex,  size_type  numChars )
  - Both of these remove the characters starting at  "startingIndex"
  - Atmost "numChars"  characters are removed.
  - If "numChars"  is not specified,  all remaining characters are removed.
  - out_of_range () is thrown if  startingIndex > size()

# Example:

- Example:
  - string s1 (“Testing”);

```
s1.erase();              // s1 is now empty.  same as  s1.clear()  or  s1 = "";
s1 = string ("Testing");       // reassign
s1.erase( 4 );                 // s1 is now "Test"
s1.erase( 1, 2 );              // s1 is now  "Tt"
s1.erase( 1, 100 );            // s1 is now  "T"
s1.erase ( 2, 100 );           // throws out_of_range()
```

# Resizing

- void string::resize( size_type num )
- void string::resize( size_type num, char c )
  - If num is greater than the string size, the string is extended and the new characters are initialized with *c*.
  - In the first flavor, new characters are initialized with NULL ('\0')

- Example:
  - ```
    string s1 ("Testing");
    s1.resize( 10 );          // s1 is 10 characters long, "Testing" + 3 NULL
    s1.resize( 3 );           // s1 is now "Tes"
    s1.resize( string::npos ); // throws length_error()
    ```
    ```

# Find

- size_type string::find ( char c ) const
- size_type string::find ( char c, size_type startingIndex ) const
  - find returns the position of the first occurrence of *c* starting at index *startingIndex* which defaults to 0 in the first case.
  - If *c* does not occur in the specified range, find returns string::npos
- Some other flavors of find are :
  - size_type string::find ( const string & str ) const
  - size_type string::find ( const string & str, size_type idx ) const
  - size_type string::find ( const char * cStr ) const
  - size_type string::find ( const char * cStr, size_type idx ) const
  - size_type string::find ( const char * cStr, size_type idx, size_type numChars ) const

# Find Example

```
string s1( "Test" );
string s2 ("st");
string s3 ("sto");

const char * charPtr = s3.c_str();

string::size_type i1 = s1.find( s2 );                    // i1  is  2
string::size_type  i2 = s1.find( s3 );                   // i2  is  string::npos

string::size_type i7 = s1.find(  charPtr, 1, 2 );        // i7  is  2
      start searching at index 1 of s1,  search for first 2 characters of charPtr

string::size_type i9 = s1.find(  charPtr, 0, 2 );        // i9  is  2
      start searching at index 0 of s1,  search for first 2 characters of charPtr

string::size_type i9 = s1.find(  charPtr, 0, 3 );        // i9  is  string::npos
      same  as  s1.find( s3 )
```

# Substrings

- string string::substr() const
- string string::substr( size_type startingIndex) const
- string string::substr( size_type startingIndex, size_type length) const

- All of these return substring of this string.
- Substring returned is substring that starts at *startingIndex,* and is of length *length.*
- If the *startingIndex* is missing, it defaults to 0.
- If *length* is missing, it means the rest of the string.

- All of these throw an out_of_range() exception if startingIndex > size()

# Iterators

- iterator string::begin()
- const_iterator string::begin() const  // for const strings

- These return an iterator "pointing" to the beginning of the string.

- iterator string::end()
- const_iterator string::end() const  // for const strings

- These return an iterator "pointing" to the end of the string (one after the last character).
- This should only be used for comparison, should not be dereferenced like *s1.end();

- Usually iterators are more useful on containers.

# Example

- string s1 (“Test”);
  string::iterator startIter = s1.begin();
  string::iterator f = find ( startIter, s1.end(), 'T');

  f points to ‘T’ in “Test”, ie, the first element at index 0

# Swap

- void string::swap ( string & str )
- void swap ( string & str1, string & str2 )

- swap will exchange the contents of the two strings.

- swap is guaranteed to have constant complexity. It is much faster than assignment, this of course is more of an issue with "big" strings.

- Also, swap can really be used in place of assignment only if you don't care about the value of the string that you are assigning from.
  - string s1 ("first"), s2("second");

    s1.swap ( s2 ); // s1 gets the value "second", but s2 gets
    // "first". If we don't care about s2 anymore,
    // then we are ok.

# Reference counting in strings

- Reference counting makes copy and assignments of strings very fast.
- This is because the string is not copied, only its reference is.
- Theoretically you need not pass const references to strings if you have reference counted strings. However, you should pass const references for portability.
- Assignments are also cheap (constant complexity).
- Since the standard does not specify implementation of string class, you need to check if the implementation that you use has reference counting.

# Input - Output for strings

- operator >> reads a string from the given input stream.

  - ```
    string  s1;
    cin >> s1;
    ```

  This will skip the leading white spaces (if *skipws* flag is set) and read all characters until it finds the next whitespace or encounters end of file or max_size() characters are read in.

- operator << outputs a string to the given output stream.

  - ```
    string  s1;
    cout << s1 << endl;
    ```

# getline

- getline is defined in the std namespace.
- It can read the input on a line by line basis.

- It ignores leading whitespace, reads all characters until the line delimiter is found.
- By default, the line delimiter is the new line character ('\n')

- Example :
  - getline ( cin, s1 ); // read a line
  - getline ( cin, s1, '*' ); // read '*' delimited strings.