H.O. #10
Fall 2015
Gary Chan

Standard Template Library (STL)

N:5,9; D:18,22

Outline

- STL
  - Vector
  - List
  - Deque
- Container adapters
  - Stack
  - Queue
- String and its operations

Consult on-line C++ tutorials for detailed usage and examples

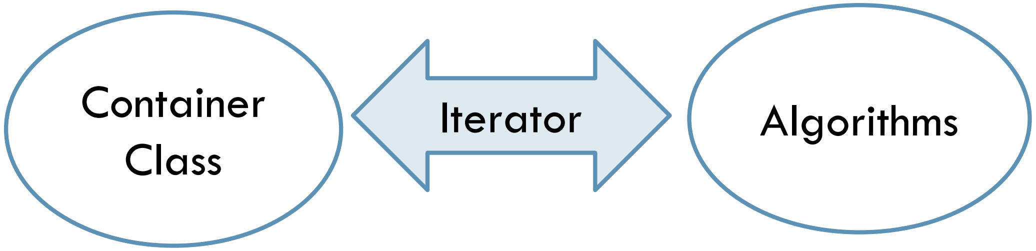
STL (Standard Template Library)

- Defines powerful, template-based, reusable components and algorithms to process them
  - Implements many common data structures
- Developed by Alexander Stepanov and Meng Lee
  - Involving many advanced C++ coding features and implementation
- Conceived and designed for performance and flexibility
- Similar interfaces between vector, list, and deque, with storage always handled transparently and automatically (expanding and contracting as needed) behind programmer's back.

STL (Standard Template Library)

Components:

- Containers:
  - Generic "off-the-shelf" class templates for storing collections of data
- Algorithms:
  - Generic "off-the-shelf" function templates for operating on containers
- Iterators:
  - Generalized "smart" pointers that allow algorithms to operate on almost any container



Containers in Standard Template Library

- Sequence containers
  - Represent linear data structures
  - Start from index/location 0
- Associative containers
  - Nonlinear containers
  - Store key/value pairs
- Container adapters
  - Implemented as constrained sequence containers
- "Near-containers" C-like pointer-based arrays
  - Exhibit capabilities similar to those of the sequence containers, but do not support all their capabilities
  - strings, bitsets and valarrays

Kind of Container	STL Containers
Sequential	vector, list, deque,
Associative	map, multimap, multiset, set
Adapters	priority_queue, queue, stack
Near-containers	bitset, valarray, string

The `vector` Container

- A type-independent pattern for an array class
  - Capacity can expand
  - Self contained
- Can be conceptualized as a powerful array
- C-style pointer-based arrays have great potential for errors and several shortcomings
  - C++ does not support continuous insertion of an elements into the array
  - Two arrays cannot be meaningfully compared with equality or relational operators (e.g., `a1 > a2`)
  - One array cannot be assigned to another using the assignment operators (e.g., `a1=a2`)

The `vector` Container

- Requires header file <vector>
- A data structure with contiguous memory locations
  - Efficient, direct access to any element via subscript operator
- Commonly used when data must be sorted and easily accessible via indices (subscripts)
- When additional memory is needed
  - Transparently allocates larger contiguous memory, copies elements and de-allocates old memory (behind user's back)
- Supports random-access *iterators*
- All STL algorithms can operate on vectors

The vector Container

- Declaration

```
template <typename T>
class vector
{  . . .  }
```

- Constructors

```
vector<int> v,              // empty vector
            v1(100),        // with capacity of 100 int
            v2(100, val),   // 100 copies of val
            v3(fptr,lptr);  // copy to v3
                            // elements in memory
                            // locations from fptr to
                            // lptr (excluding lptr)
```

```
int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
vector<int> v1( &array[2], &array[4]);//gets 3 and 4
```

Vector Operations

- Information about a vector's contents
  - `v.size()        // current # of items`
  - `v.empty()`
  - `v.capacity()    // max. storage space(no less than v.size())`
  - `Etc.`
- Adding, removing, accessing elements
  - `v.push_back(X)   // push as back`
  - `v.pop_back()     // take away the back`
  - `v.front()        // peep the front`
  - `v.back()         // peep the back`
- Declaring different types of vectors:
  - `vector<int> iv;  // empty integer vector`

Vector Operations

- Assignment
  - v1 = v2
- Swapping
  - v1.swap(v2)
- Relational operators
  - == or != implies element by element equality or inequality
  - less than <, <=, >, >= behave like string comparison
- Accessing an element
  - With [ ]: E.g., v[0], v[1], etc.
  - With the member function at(i): E.g., v.at(0), v.at(1), etc.
- The member function `at()` has boundary checking:

```
vector<int> iv;  // empty vector of size 0

for( i = 0; i < 10; i++ ){
   cout << iv[i]; // segmentation fault (due to out-of-range access)
   cout << iv.at(i); // graceful termination with an exception msg:
   //terminate called after throwing an instance of 'std::out_of_range'
  }
```

Increasing Capacity of a Vector

- When vector **v** becomes full
  - Capacity is increased automatically when item is added
- Algorithm to increase capacity of **vector<T>**
  - Allocate new array to store vector's elements
  - Copy existing elements to new array
  - Destroy old array in **vector<T>**
  - Make new array the **vector<T>**'s storage array
- Allocate new array
  - Capacity doubles when more space is needed
    - 0 → 1 → 2 → 4 → 8 → 16, etc.
  - Can be wasteful for a large vector to double – use `resize()` to resize the vector, e.g.,

```
v.resize( 10 ); // the elements beyond the size will be
                // truncated/erased
```

2-D vector

- Accessing element as `v[i][j]`
- Creating a 100x1000 matrix. Method 1:

```
  int i, j;
  vector<vector<int> > v2D; // Note the space between > >

// creating a 1000x100 matrice
  for(i = 0; i < 1000; i++){
    v2D.push_back( vector<int> () ); // a row element
    for(j=0; j<100; j++)
      v2D[i].push_back(i+j);  // pushing column elements
  }
```

- Method 2:

```
vector<vector<int> > v2d;
v2d.resize(1000);
for( i = 0; i < 1000; i++ )
  v2d[i].resize(100);
```

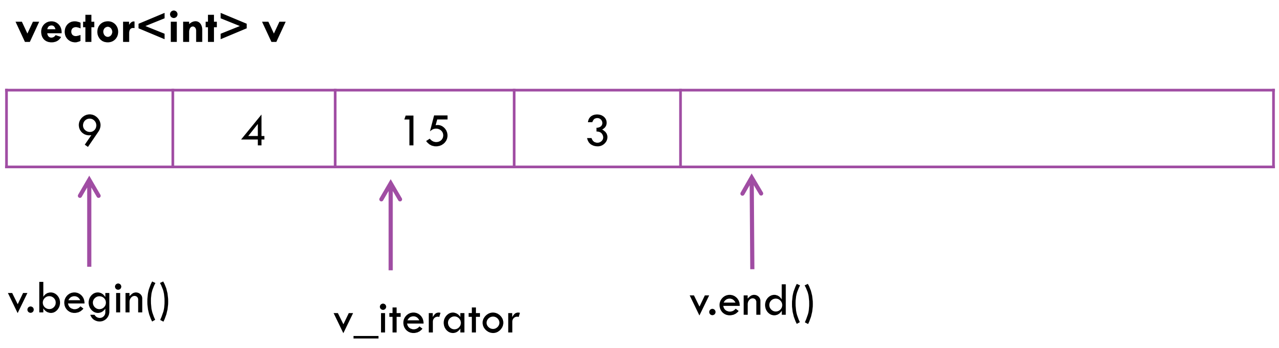
Iterators

- Note that a subscript operator is provided for vector, e.g., v[2]
  - BUT … this is not a *generic* way to access container elements
  - This is because some containers do NOT have `[]` connotations, and hence their `[]` operator is not overloaded (list, etc.)
- STL provides objects called iterators

  -
    ```
    vector<int>::iterator foo;
    vector<int>::const_iterator foo;
    vector<int>::reverse_iterator foo;
    vector<int>::const_reverse_iteractor foo;
    ```
  - can point at an element
  - can access the value within that element
  - can move from one element to another
- They are independent of any particular container … thus a generic mechanism as a uniform way to access elements
- A constant iterator is an iterator which you will not or cannot change the content it points to

Iterators

- Given a vector which has had values placed in the first 4 locations:



vector<int> v

9	4	15	3	

v.begin()

v_iterator

v.end()

- **`v.begin()`** will return the iterator value for the first slot
- **`v.end()`** for the next empty slot
- `for( v_iterator = v.begin(); v_iteractor < v.end(); v_iterator++)…`

Iterators

- Each STL container declares an `iterator` type
  - can be used to define `iterator` objects
- To declare an `iterator` object, the identifier `iterator` must be preceded by
  - name of container, e.g., `vector<int>`
  - scope operator `::`
- Example:

```
vector<int>::iterator vecIter = v.begin()
 vector<int>::cont_iterator cvecIter = v.begin()
```

Iterators

- A pointer
- Basic operators that can be applied to iterators:
  - Increment operator ++
  - Decrement operator --
  - Dereferencing operator *
  - Assignment =
  - Addition, subtraction +, -, +=, -=

    `vecIter + n` returns iterator positioned `n` elements away
  - Subscript operator [ ]

    `vecIter[n]` returns reference to nth element from current position

Iterators vs. Subscript for Vector

Subscript:

```
ostream & operator<<(ostream & out, const
  vector<double> & v)
{
  for (int i = 0; i < v.size(); i++)
    out << v[i] << "  ";
  return out;
}
```

Iterators:

```
for (vector<double>::iterator it = v.begin();
  it != v.end(); it++)   // can also it < v.end()
      out << *it << "  ";
```

Iterator Functions

- Insert and erase elements anywhere in the vector with iterators is as inefficient as for arrays because shifting is required

Function Member	**Description**
v.begin()	Return an iterator positioned at *v*'s first element
v.end()	Return an iterator positioned past *v*'s last element
v.rbegin()	Return a reverse iterator positioned at *v*'s last element
v.rend()	Return a reverse iterator positioned before *v*'s first element
v.insert(iter, value)	Insert *value* into *v* at the location specified by *iter*
v.insert(iter, n, value)	Insert *n* copies of *value* into *v* at the location specified by *iter*
v.erase(iter)	Erase the value in *v* at the location specified by *iter*
v.erase(iter1, iter2)	Erase values in *v* from the location specified by *iter1* to that specified by *iter2* (*not including* *iter2*)
+ other insert /erase overload functions	

Iterator does not move with vector; Need to be re-located

```
#include <iostream>
#include <vector>
using namespace std;

int main(){

  vector<int> v(2,1);  // two 1s
  vector<int>::iterator vit, it;
  int a[] = {1, 2, 3, 4, 5};

  vit = v.begin();

  cout << "v address = " << v.begin() << "; content = " << v[0] << endl;
  cout << "vit address = " << vit << "; content = " << *vit << endl;

  v.insert( vit, a, a+4 );
  for( it = v.begin(); it < v.end(); it++ )
    cout << *it << " ";
  cout << endl;

  cout << "v address = " << v.begin() << "; content = " << v[0] << endl;
  cout << "vit address = " << vit << "; content = " << *vit << endl;
  return 1;
}
```

```
v address = 0x3fcc8; content = 1
vit address = 0x3fcc8; content = 1
1 2 3 4 1 1
v address = 0x3fd68; content = 1
vit address = 0x3fcc8; content = 261328
```

Template Function and Its Call

```
#include <iostream>
#include <vector>

using namespace std;

template< class A>
void printv( vector< A > a ){
  typename vector< A >::const_iterator it;
  // need typename here as A is a template

  for( it = a.begin(); it < a.end(); it++ )
    cout << *it << " ";
  cout << "\n";
}

template< class A, class B>
void print2( void ){
  A a = 3;
  B b = "hi there";

  cout << a << "\n";
  cout << b << "\n";

}
```

```
int main(){
  vector<int> vint;
  int i;

  for( i = 0; i < 10 ; i++ )
    vint.push_back( i );
  printv( vint );

  print2<int, char *>();

  return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
3
hi there
```

Some Common Vector Member Functions

- vector::assign
  - Assign values to vector
- vector::at(i)
  - ith element of the vector (start at 0)
- vector::back()
  - The reference of the last element
- vector::begin()
  - The first element for iterator
- vector::capacity()
  - Storage capacity of the vector
- vector::clear()
  - Clear the content
- vector::empty()
  - Whether the vector is empty
- vector::end()
  - The last element for iterator
- vector::erase
  - Remove elements
- vector::front()
  - Return the reference to the first element
- vector::insert
  - Insert elements into the vector
- vector::operator[]
  - foo[i] is the ith element of the vector
- vector::pop_back()
  - pop out the last element
- vector::push_back(X)
  - Push X as the last element
- vector::rbegin()
  - Reverse begin for iterator
- vector::rend()
  - Reverse end for iterator
- vector::size()
  - The number of elements
- vector::swap(v2)
  - v1.swap(v2) swaps v1 and v2

vector.cpp and its Sample Output

```
3 4
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

- The vector’s capacity increases to accommodate the growing size
- Note how to print a vector using template and its iterator implementation:

```
template < typename T > void printVector( const vector< T > &);
```

Vectors vs. Arrays

Vectors

- Capacity can increase
- A self-contained object having function members to do tasks
- Is a class template

(Primitive) Arrays

- Fixed size, cannot be changed during execution
- Cannot "operate" on itself: must write functions to work on it
- Must "re-invent the wheel" for most actions for each array element type

STL's `list` Container

- Requires header file `<list>`
- Implemented internally as a doubly-linked list
  - Provides efficient insertion and deletion operations at any location
- Supports bidirectional iterators
  - Can be traversed forward and backward

Creating a vector of list

```
#include <vector>
#include <list>
#include <iostream>
using namespace std;

int main(){

  vector<list<int> > vl(10); // remember the space between > >
  int i,j;

  for( i = 0; i < 10; i++)
    for(j = 0; j<5; j++)
      vl[i].push_back(j); // create a vector of identical lists

  list<int>::reverse_iterator lit = vl[3].rbegin();
  for( ; lit != vl[3].rend(); lit++ )
    cout << *lit;    // print out 43210

  return 1;
}
```

Converting a reverse iterator to a normal iterator

- Note that the forward iterator is one position ahead of the reverse iterator
  - `it.begin()` points to the first element, while `rit.rend()` points to the position just before the first one
  - `it.end()` points to the next one after the last element, while `rit.rbegin()` points to the last one.
- It is often useful to convert iterator ←→ reverse_iterator
- Iterator → reverse_iterator: use the constructor of the reverse_iterator, e.g.,

```
list<int>::reverse_iterator rit( it );
```

- Reverse_iterator → iterator: use the `base()` member function in the reverse_iterator, e.g.,

```
it = rit.base();
```

- In all the cases, the iterator after the conversion is always one position higher than the reverse_iterator.

```
int main(){

  list<int> coll;

  // insert elements from 1 to 9
  for (int i=1; i<=9; ++i) {
    coll.push_back(i);
  }

  // find position of element with value 5
  list<int>::iterator pos;
  pos = find (coll.begin(), coll.end(),    // range
              3);                          // value

  // print value of the element
  cout << "pos:   " << *pos << endl;  // get 3

  // convert forward iterator to reverse iterator using its constructor
  list<int>::reverse_iterator rpos(pos);

  // print value of the element to which the reverse iterator refers
  cout << "rpos:  " << *rpos << endl; // get 2!

  // convert reverse iterator back to normal iterator
  list<int>::iterator rrpos;
  rrpos = rpos.base();

  // print value of the element to which the normal iterator refers
  cout << "rrpos: " << *rrpos << endl; // get 3


}
```

list Member Function `sort()`

- By default, arranges the elements in the list in ascending order
- Can take a binary predicate (i.e., boolean) function as argument to determine the sorting order
  - Called like a function pointer
  - E.g., `mylist.sort(compare_alg),` where `compare_alg(x,y)` returns true if x is ordered *before* y.

```
// list::sort
#include <iostream>
#include <list>
using namespace std;

// reverse sort (sort in decreasing order)
bool reverse_sort (int left, int right)
{
  if (left > right)
    return true;   // first comes before second

  return false;
}

int main ()
{
  list<int> lst;
  list<int>::const_iterator it;

  lst.push_back( 2 );
  lst.push_back( 1 );
  lst.push_back( 7 );
  lst.push_back( 9 );
  lst.push_back( 6 );
  lst.push_back( 2 );
```

```
  lst.sort();

  cout << "lst sorted in increasing order:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  lst.sort(reverse_sort);

  cout << "lst sorted in decreasing order:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```

```
lst sorted in increasing order: 1 2 2 6 7 9
lst sorted in decreasing order: 9 7 6 2 2 1
```

list Member Function `unique()`

- Removes duplicate elements from the list
- List must first be *sorted*
- Can take an argument which specifies a binary predicate (i.e., boolean) function to determine whether two elements are *equal*
  - Called like a function pointer
  - Scanning the list from the head and compare the most recently retained element with a new one. Delete the new one if it is "the same" as the retained one.
  - Define an equal function, say `bool equal(x,y)` , which returns true if `x` is defined to be equal to `y`. In the context of list, `x` is the retained element right before `y`. Then a call of

    `unique( equal )` removes `y` if `equal` returns true, and not otherwise.

```
// list::unqiue
#include <iostream>
#include <list>
using namespace std;
// definition of equal
// if left is less than or equal to a factor 2 of right, they are the same
// left is always before right in the list and they are +ve integers
bool factor2 (int left, int right)
{
  cout << compare << " " << remove
       << endl;

  if (left *2 > right){
    cout << "true!\n"; // equal
    return true; // delete remove
  }
  return false;
}

int main (){
  list<int> lst;
  list<int>::const_iterator it;

  lst.push_back( 2 );
  lst.push_back( 1 );
  lst.push_back( 7 );
  lst.push_back( 9 );
  lst.push_back( 3 );
  lst.push_back( 2 );
```

```
  lst.sort();
  lst.unique();
  cout << "lst after unique call:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  lst.unique(factor2);
  cout << "lst after unique(factor2) call:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```

```
lst after unique call: 1 2 3 7 9
1 2
2 3
true!
2 7
7 9
true!
lst after unique(factor2) call: 1 2 7
```

Some list Member Functions

- list::assign
- list::back()
- list::begin()
- list::clear()
- list::empty()
- list::end()
- list::erase()
- list::front()
- list::insert
- list::merge
  - v1.merge(v2) merges the two *sorted* lists to form a new sorted list v1
- list::operator=
- v1=v2
- list::pop_back()
- list::pop_front()
- list::push_back(X)
- list::push_front(X)
- list::rbegin()
- list::remove()
- list::remove_if(foo)
  - Remove all elements for the function foo returning true
- list::rend()
- list::reverse()
  - Reverse the order of the list
- list::size()
- list::sort(foo)
  - Sort the element of the list
- list::swap(list2)
  - Swap the two lists
- list::unique(foo)
  - Remove all the duplicates in a *sorted* list

list.cpp and its Sample Output

- Note the print template of list and the iterator:

```
template < typename T > void printList( const list<T> & );
```

- Output:

```
values contains: 2 1 4 3 1

values after sorting contains: 1 1 2 3 4

After unique, values contains: 1 2 3 4

After remove( 4 ), values contains: 1 2 3
```

STL's deque Container

- Requires header file <deque>
- As an ADT, a deque is a double-ended queue
  - Pronounced as "deck"
- It is a sequential container
  - Additional storage may be allocated at either end
  - Noncontiguous memory layout (dynamic allocation on the heap)
- Acts like a queue (or stack) on both ends
- It is an ordered collection of data items
- Items usually are added or removed at the ends
- Provides many of the benefits of vector and list in one container
  - Reasonably efficient indexed access using subscripting
  - Reasonably efficient insertion and deletion operations at front and back

Deque Operations

- Construct a deque (usually empty)

```
deque<double> dq;         // empty deque

deque<int> first (3,100); // three ints with a value of 100
deque<int> second (5,200); // five ints with a value of 200
```

- *Empty*: return true if the deque is empty
- Add
  - *push_front*: add an element at the front of the deque
  - *push_back*: add an element at the back of the deque
- Retreive
  - *front*: peep the element at the front of the deque. Can be lvalue.
  - *back*: peep the element at the back of the deque. Can be lvalue.
- Remove
  - *pop_front*: remove the element at the front of the deque
  - *pop_back*: remove the element at the back of the deque

Deque Class Template

- Has the same operations as `vector<T>` except some member functions (there is no `capacity()` and no `reserve()`)
- Has two new operations:
  - `d.push_front(value);` - Push copy of value at `front` of `d`
  - `d.pop_front();` - Remove the element at the front of `d`
- Like STL's `vector`, it has
  - [ ] subscript operator
  - insert and delete at arbitrary points in the list (*insert* and *erase*)
- Insertion and deletion in the middle of the deque are not guaranteed to be efficient

Some deque Member Functions

- deque::assign
- deque::at(i)
  - The ith element (starting from 0)
- deque::back()
  - Return the last element
- deque::begin()
- deque::clear()
  - Delete the whole deque
- deque::empty()
- deque::end()
- deque::erase
  - Remove either a single element (erase(i)) or a range of element (erase(i,j))
- deque::front()
  - Return the first element
- deque::insert
- deque::operator=
  - for d1 = d2;
- deque::operator[]
  - for d[i]
- deque::pop_back()
  - delete the last element
- deque::pop_front()
  - delete the first element
- deque::push_back(X)
- deque::push_front(X)
- deque::rbegin()
- deque::rend()
- deque::size()
  - Return the number of elements
- deque::swap(dq2)

deque.cpp and Sample Output

```
Contents after alternating adds at front and back:
5  3  1  2  4  6
Contents (via iterators) after changing back and popping front:
3  1  2  4  999
Dumping the deque from the back:
999  4  2  1  3
```

Efficiency Consideration and Performance Comparison

- Which STL to use depends on the access pattern of your applications
- Their insertion and deletion are all through iterators
- Vector (Implemented as a contiguous array)
  - `insert` and `erase` in the middle of vector are not efficient (involves moving of elements and may lead to memory re-allocation and copying)
  - Insertion and deletion at the *end* are fast (e.g., `push_back` operation)
  - Random *access* is fast (array indexing, e.g., `front`, `back` and `[]`)
- List (Implemented as doubly linked list)
  - `insert` and `erase` in the middle of the list given an iterator are efficient (involving only a few pointer movements)
  - Insertion and deletion at *both ends* are fast (`push_front` and `push_back` operations)
  - Random access is slow (has to use iterator to traverse the list to the get the element)
- Deque
  - Implementation involves a combination of pointers and array (blocks of contiguous memory chunks), probably in the form of a linked list with array in each node
  - `insert` and `erase` in the middle are reasonably fast
  - Insertion and deletion at both ends are reasonably fast (`push_front` and `push_back` operations)
  - Random access is reasonably fast (using `[]`)
  - Intermediate performance between vector and list

Container Adapters

Container Adapters

- Are not first-class containers
  - Do not provide the actual data structure implementation
  - Do not support iterators
- You can choose an appropriate underlying data structure
  - list, deque, vector
- With some useful member functions
  - Push: properly insert an element into data structure
  - Pop: properly remove an element from data structure

STL's stack Adapter

- STL `stack` container
  - Actually an adapter
  - Indicated by container type `C<T>` as one of its type parameters

    `stack<T, C<T> > aStack;`
- If no container specified `stack<T> astack;`
  - Default is `deque`
  - Also possible to specify a `list` or `vector` as the container for the stack
- Enables insertions and deletions at one end
  - Last-in first-out (LIFO) data structure

STL's Stack Adapter

- Requires header file <stack>
- Operations (call functions of the underlying container)
  - push – insert element at top (calls push_back)
  - pop – remove top element (calls pop_back)
  - top – returns reference to top element (calls back)
  - empty – determine if the stack is empty (calls empty)
  - size – get the number of elements (calls size)
- The implementation of these functions are straightforward given the underlying classes
- Each common operation is implemented as an `inline` function
  - For the function caller to directly call the appropriate function of the underlying container
  - Avoid the overhead of a second function call

stack.cpp and its Sample Output

```
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

STL's queue Adapter

- A `queue` can be specified

  `queue<T, C<T> > aQueue;`

  - `C<T>` may be any container supporting `push_back()` and `pop_front()`
- The default container is `deque`
  - Could also use `queue<T, list<T> > aQueue;`
  - For the best performance, use class `deque` as the underlying container
- Enables insertions at back and deletions from front
  - First-in first-out (FIFO) data structure

STL's queue Adapter

- Operations (call functions of the underlying container)
  - push – insert element at back (calls push_back)
  - pop – remove element from front (calls pop_front)
  - front – returns reference to first element (calls front)
  - empty – determine if the queue is empty (calls empty)
  - size – get the number of elements (calls size)
- Again, each common operation is implemented as an inline function

STL Demonstration

- queue.cpp and its Output:

```
Popping from values: 3.2 9.8 5.4
```

- STL.cpp

C++ String Class

Outline

- String Initialization
- Basic Operations
- Comparisons
- Substrings
- Swapping Strings
- String Size
- Finding Strings and Characters
- Replacing Characters
- Inserting Characters
- String Stream

Strings

- Strings are a special data type in C++ used to store a sequence of characters
- Include the <string> library to use strings
- Compare strings with the <, ==, and != operations
- Concatenate strings with the + operation
- Use `s.substr(position, size)` to get a substring from a string `s` starting from `position`, and of length `size`
  - Starting position is 0
- Use `s.find(subs)` to find where a substring `subs` begins in string `s`

The C++ String Class

- Always from index/location 0
- Contains all valid characters: has kept track of NULL characters behind user's back
- Variety of constructors provided for defining strings
  - Define an empty string

    ```
    string s;
    ```
  - Define a string initialized with another string

    ```
    string s("some other string”);
    ```

string s;	Constructs *s* as an empty string
string s(str_ca);	Constructs *s* to contain a copy of string or char array *str_ca*
string s(ca, n);	Constructs *s* to contain a copy of the first *n* characters in char array *ca*
string s(str, pos, n);	Content is initialized to a copy of the tail part of the string *str*. The substring copied is the portion of *str* that begins at the character position *pos* and takes up to *n* characters (it takes less than *n* if the end of *str* is reached before). To copy till the end of *str*, *n* can be set to be very large, string::npos, str.length() or simply omitted, e.g., `s( s1, 4, string::npos)` or `s( s1, 4, s1.length())` or `s( s1, 4)`
string s(n, ch)	Constructs *s* to contain *n* copies of the character *ch*

The C++ String Initialization

- Creates an empty string containing no characters

```
string empty;
```

- Creates a string containing the characters "hello”

```
string text( "hello" );
```

- Creates a string containing eight 'x' characters

```
string name( 8, 'x' );  //MUST use single-quote
```

- Creating a string from a substring:

```
char * cptr = "BCDEF\0"; // character array
string s1 (cptr, 2); // get BC, treated as ca
string s2 (“hixyz”, 3); // get hix, treated as ca
string s3 ( s2, 2 ); // get xyz, treated as str and get
                     // from index 2 to the end
```

- Implicitly performs `string month( "March" );`

```
string month = "March";
```

The C++ String Class

- No conversion from int or char in a string definition
  - Wrong statement (produce syntax errors)

```
string error1 = 'c';  // use string s1 = “c” or s1(1,’c’);
string error2( 'u' ); // use string s2( “u” );
string error3 = 22;   // use string s3( “22”);
string error4( 8 );
```

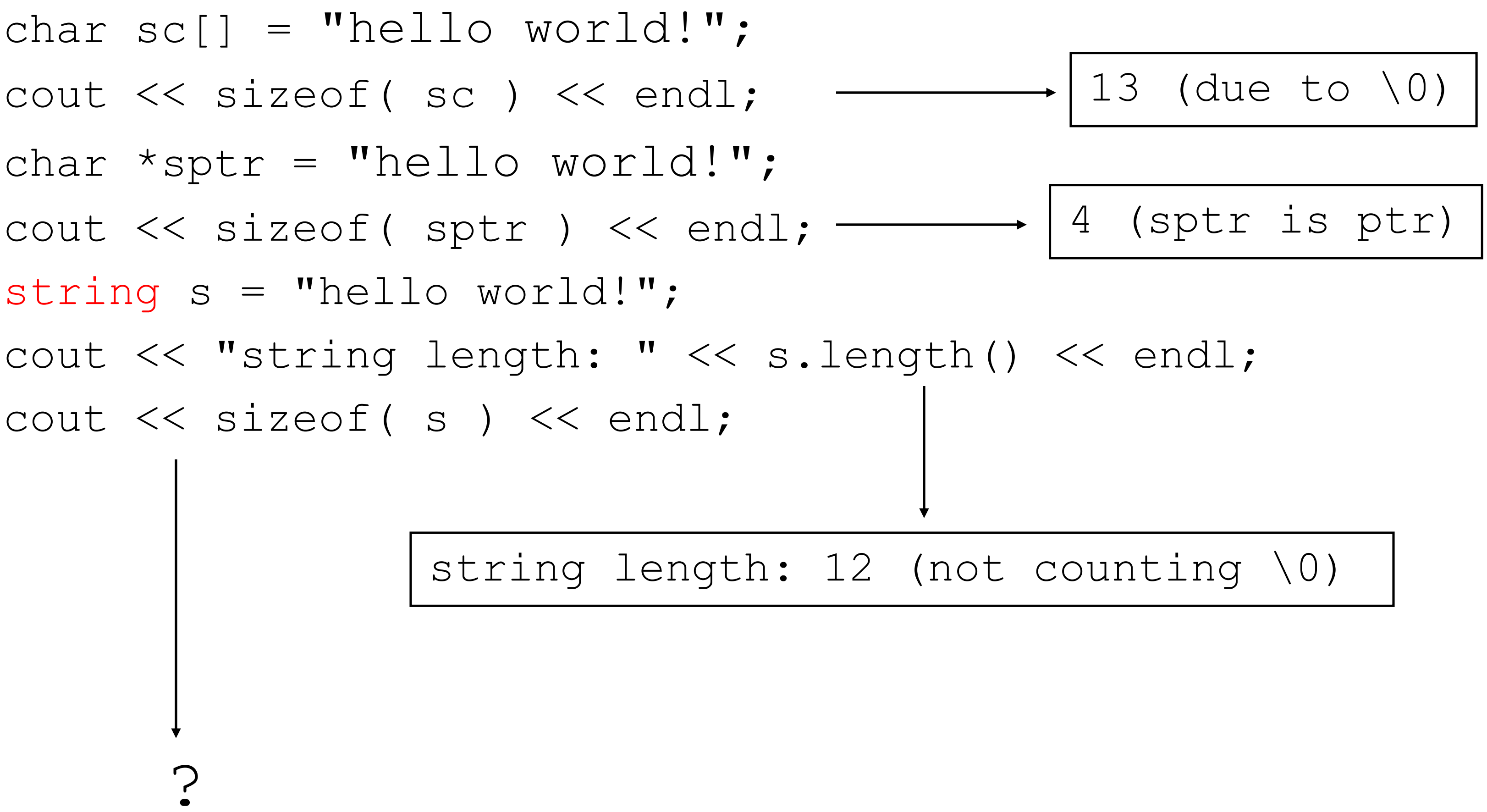
- Assigning a single character to a string object is allowed
  - Example

```
string1 = 'n'; // this is NOT constructor and hence ok
string2 = “n”; // ok also
```

Basic String Operations

- Length of string: Exclude \0 character
  - `length()`
- Input and output
  - Use insertion `<<` and extraction `>>` operators
    - Input delimited by white-space characters
  - `getline(cin, str)` for reading a string and including white spaces till a newline (‘\n’)
    - Note that `cin >> str`; will NOT get the whole line, but only ONE word right before the space

Some string commands



```
char sc[] = "hello world!";
cout << sizeof( sc ) << endl;
char *sptr = "hello world!";
cout << sizeof( sptr ) << endl;
string s = "hello world!";
cout << "string length: " << s.length() << endl;
cout << sizeof( s ) << endl;
```

13 (due to \0)

4 (sptr is ptr)

string length: 12 (not counting \0)

?

C++ string is a Class Object

- The result is
  - 16 with WinXP VC++
  - 8 with Linux g++
  - 4 with Unix g++
- `sizeof(a_class)` returns total size of the class object, including its member variables with stuffing bytes, plus size of VMT (virtual method table), if virtual functions are involved
  - If you don’t understand virtual functions in C++, forget about the last half of the sentence
  - Stuffing policy is complicated and partly depends on the compiler
  - The result of 4 in Unix g++ may be due to the use of a pointer, while the other results may be some other variables in the class

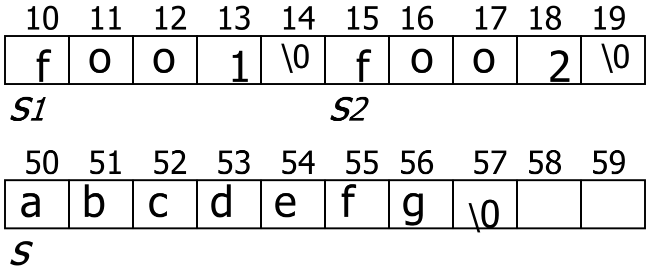
Strings

- Built-in C-style strings are implemented as an array of characters.
- Each string ends with the special null-terminator ‘\0’.
- *strcpy*: used to copy strings

  strcmp: used to compare strings

  strlen: used to determine the length of strings
- Individual characters can be accessed by the array indexing operator

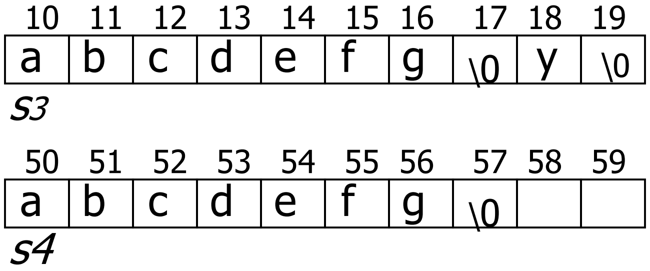
```
char s1[] = “foo1”;

char s2[] = “foo2”;

char s[]=“abcdefg”;
```



10	11	12	13	14	15	16	17	18	19
f	o	o	1	\0	f	o	o	2	\0
s1					*s2*				

50	51	52	53	54	55	56	57	58	59
a	b	c	d	e	f	g	\0		
s									

```
strcpy(s3, s4);
//copy s4 to s3
//(s3 must have enough size)
//including \0
```



10	11	12	13	14	15	16	17	18	19
a	b	c	d	e	f	g	\0	y	\0
s3									

50	51	52	53	54	55	56	57	58	59
a	b	c	d	e	f	g	\0		
s4									

Strings: Example 1

```
#include <iostream>
#include <string>       // string library
using namespace std;

int main(){
  string name;
  cout << "Enter name (without spaces): ";
  cin >> name;
  cout << "Name: " << name << endl;
}
```

- **example input/output:**

```
Enter name (without spaces): GaryChan
Name: GaryChan
```

String Operations

- String concatenation

s + str_ca_ch	Returns the result of concatenating s and string/char-array/character *str_ca_ch*
s.append(str_ca)	Appends *str_ca* at the end of *s*; returns *s*
s.append(ca, n)	Appends the first *n* characters in *ca* at the end of *s*; returns *s*
s.append(n, ch)	Appends *n* copies of *ch* at the end of *s*; returns *s*

```
string s;
s = "A";
char * cptr = "BCDE\0";
s += cptr;  // s is now ABCDE
cout << s << s.length() << endl; // s’s length is 5
```

String Copy

- Operators `=` and `+=` make copies of part or all of a string
  - `s1 = s2;`
  - `s1 += s2;`
- Function assign()
  - copies the contents of a string into another string

    ```
    s.assign(s2);  // same as s = s2;
    ```

  - copies a specified range of characters

    ```
    s.assign(sourceString,start_index,numberOfCharacters);
    ```

Accessing Individual Characters

- Use overloaded subscript operator `[ ]`
- First subscript is 0, last subscript is length() – 1 (No need to worry about NULL character)
    - Note that you CANNOT set str[i] where i is not within the string length. The operation will be ignored.
- Function at()
    - Like `[ ]`, but provides checked access (or range checking)
    - Error message (with an exception) if the index is beyond the string length

string1.cpp and its Sample Output

```
string1: cat
string2: cat
string3: cat
After modification of string2 and string3:
string1: cat
string2: rat
string3: car (at() function demonstrated)
After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb
comb4
```

String Comparisons

- Comparisons
  - Overloaded operators for `<`, `<=`, `>`, `>=`, `==`, `=!`
- Also `compare()` function
  - s1.compare(s2): Returns 0 if the strings are equivalent
  - Returns positive number if the current string `s1` is lexicographically greater than the argument string `s2` (i.e., `"T" < "t"; "a" < "abc"`)
  - Returns negative number if the current string is lexicographically less than the argument string

string2.cpp Sample Output

```
string1: Testing the comparison functions.
string2: Hello
string3: stinger
string4: Hello

string1 > string4
string1.compare( string2 ) > 0
string1.compare( 2, 5, string3, 0, 5 ) == 0
string4.compare( 0, string2.length(), string2 ) == 0
string2.compare( 0, 3, string4 ) < 0
```

Substrings

- Retrieves a substring from a string
  - Returns a new string object copied from the source string
- First argument
  - Specifies beginning subscript of desired substring
- Second argument
  - Specifies length of desired substring

string3.cpp

```
int main()
{
   string string1( "The airplane landed on time." );

   // retrieve substring "plane" which
   // begins at subscript 7 and consists of 5 characters
   cout << string1.substr( 7, 5 ) << endl;

   return 0;
} // end main
```

- Sample Output

```
plane
```

Swapping Strings

- Swaps contents of the current string and the argument string
- Useful for implementing programs that sort strings

s.swap(str)
swap(s, str)
	Swaps the contents of *s* and *str*; return type is *void*

string4.cpp

```
int main()
{
   string first( "one" );
   string second( "two" );

   cout << "Before swap:\n first: " << first
        << "\nsecond: " << second;

   first.swap( second ); // swap strings

   cout << "\n\nAfter swap:\n first: " << first
        << "\nsecond: " << second << endl;
   return 0;
} // end main
```

- Swap the values of first and second

```
Before swap:
 first: one
second: two
After swap:
 first: two
second: one
```

String Size

- Capacity: capacity()
  - Number of characters that can be currently stored without allocating more memory
    - Must be at least equal to the size() (or its alias length()), can be greater
    - Depends on the implementation
- Maximum size: max_size()
  - Largest possible size a string can have
    - If exceeded, a length_error exception is thrown
- Member function empty()
  - Returns true if the string is empty
- Member function resize()
  - Changes the valid *length* of the current string
  - If the resize number is larger than the current size, additional elements are set to some character as specified in the second argument (default to NULL character), i.e., include the tail characters into the string
  - If the resize number is smaller than the current size, the string is truncated

String Size Operations

s.capacity()	Returns the capacity of the storage allocated in *s*

s.size()
s.length()
	Returns the length of *s*

s.empty()	Returns *true* if *s* contains no characters, *false* otherwise
s.max_size()	Retruns the largest possible capacity of *s*
s.resize(n, ch='\0')	Resize the string s to size n, by adding ch (default to NULL) characters if n is larger than string size

string5.cpp and its Sample Output

```
Statistics before input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 4294967293
size: 6
length: 6
empty: false

The remaining string is: soup
capacity: 15
max size: 4294967293
size: 4
length: 4
empty: false
```

```
string1 is now:
soup1234567890abcdefghijklmnopqrstuvwxyz123
4567890
capacity: 63
max size: 4294967293
size: 50
length: 50
empty: false

Stats after resizing by (length + 10):
capacity: 63
max size: 4294967293
size: 60   (resized, padded with NULL char)
length: 60 (resized, padded with NULL char)
empty: false
```

Finding Strings and Characters

- Return -1 is not found
- `find`
  - Attempts to find specified string in the current string
  - Returns starting location of the string at the *beginning* of the matched string if found
  - Returns the value `string::npos` otherwise
    - All string find-related functions return this const static value to indicate the target was not found
- `rfind`
  - Searches current string backward (right-to-left) for the specified string
  - If the string is found, its subscript location at the *end* of the matched string is returned

Finding Strings and Characters

- `find_first_of`
  - Locates first occurrence in the current string of any character in the specified string
- `find_last_of`
  - Locates last occurrence in the current string of any character in the specified string
- `find_first_not_of`
  - Locates first occurrence in the current string of any character not contained in the specified string

String Search Accessors

s.find(str_ca_ch, pos)	Returns the first position >= *pos* such that the *returned position* matches the beginning of *str_ca_ch*; returns *npos* if there is no such position; 0 is the default value for *pos*
s.find_first_of(str_ca_ch, pos)	Returns the first position >= *pos* of *a character* in *s* that matches *any character* in *str_ca_ch*; returns *npos* if there is no such position; 0 is the default value for *pos*
s.find_first_not_of(str_ca_ch, pos)	Returns the first position >= *pos* of a character in *s* that does *not* match *any of the character* in *str_ca_ch*; returns *npos* if there is no such position; 0 is the default value for *pos*
s.find_last_of(str_ca_ch, pos)	Returns the highest position <= *pos* of a character in *s* that matches any character in *str_ca_ch*; returns *npos* if there is no such position; *npos* is the default value for *pos*
s.find_last_not_of(str_ca_ch, pos)	Returns the highest position <= *pos* of a character in *s* that does *not* match *any character* in *str_ca_ch*; returns *npos* if there is no such position; *npos* is the default value for *pos*

string6.cpp and its Sample Output

```
Original string:
noon is 12 pm; midnight is not.
0123456789012345678901234567890

(find) "is" was found at: 5
(rfind) "is" was found at: 25

(find_first_of) found 'o' from the group "misop" at: 1

(find_last_of) found 'o' from the group "misop" at: 28

(find_first_not_of) '1' is not contained in "noi spm" and was
found at:8

(find_first_not_of) ';' is not contained in "12noi spm" and was
found at:13

find_first_not_of("noon is 12 pm; midnight is not.") returned: -1
```

Strings: Example 2

```
#include <iostream>
#include <string>       // string library
using namespace std;

int main(){
  string s = "Top ";
  string t = "ten ";
  string w;
  string x = "Top 10 Uses for Strings";

  w = s + t;
  cout << "s: " << s << endl;
  cout << "t: " << t << endl;
  cout << "w: " << w << endl;
  cout << "w[5]: " << w[5] << endl;
```

Strings: Example 2

```
  if(s < t)
      cout << "s alphabetically less than t" << endl;
  else
      cout << "t alphabetically less than s" << endl;
  if(s+t == w)
      cout << "s+t = w" << endl;
  else
      cout << "s+t != w" << endl;

  cout << "substring where: " << x.find("Uses") << endl;
  cout << "substring at position 12 with 7 characters: "
       << x.substr(12, 7) << endl;
  return 0;
}
```

Strings: Example 2

- **Example output:**

```
s: Top
t: ten
w: Top ten
w[5]: e
s alphabetically less than t
s+t = w
substring where: 7
substring at position 12 with 7 characters: for Str
```

- What about `x.find("None")`?
- If the substring is not found, `find` returns a number that is larger than any legal position within the string.

More String Examples

```
 string s = "hello world!";
char sc[] = "hello world!";
cout << sizeof( s ) << endl;
cout << "string length " << s.length() << endl;
cout << "string length " << sizeof( sc ) << endl;
cout << s.find("xyz") << endl;

string x1 = "foo";
string x2 = x1;
cout << "x1="<< x1 << endl;
cout << "x2="<< x2 << endl;
x2 = "bar";  // resize string x2
cout << "x1="<< x1 << endl;
cout << "x2="<< x2 << endl;
```

```
4
string length 12
string length 13
4294967295
x1=foo
x2=foo
x1=foo
x2=bar
```

Extract away white spaces and output the first integer in the string

```
cssu5:~> a.out
fjdlasfjlj   fjljaflsja
fjdlasfjljfjljaflsja(Space-eliminated string)
Please input a line starting with integer
cssu5:~> a.out
jfdljaf jfld123  456 jfdljasf
jfdljafjfld123456jfdljasf(Space-eliminated string)
123456
cssu5:~> a.out
jfdlasjfd 123  fjldja j234jfdljasf
jfdlasjfd123fjldjaj234jfdljasf(Space-eliminated string)
123
```

```
// readstring.cpp
// simple program to read in a line and eliminate all the white spaces
// Echo those non-space characters
// Print out the first integer found in the string

#include <iostream>
#include <cctype>
#include <cstring>
#include <stdio.h>      // for sscanf()

using namespace std;

#define MAX_LINE 100   // maximum character in a line

main(){
  char ch;
  char str[ MAX_LINE ];  // the string to eliminate white spaces from
  int write_pos = 0;     // index into str
  int i;
  int number;
  // get rid of all the spaces in a line and
  // output only those non-space characters
  while( (ch = cin.get()) != '\n' )
    if( !isspace( ch ) )
      str[ write_pos++ ] = ch;

  str[ write_pos ] = '\0';      // putting end of line character
  cout << str << " (Space-eliminated string)" << endl;

  // finding the first integer in the string
  for( i = 0; i < write_pos; i++ )
    if( isdigit( str[i] ) )
      break;

  if( i == write_pos )
    cout << "Please input a line starting with integer\n";
  else{
    sscanf( &str[i], "%d", & number );  //scanning the first integer
    cout << number << endl;
  }

  return 0;
}
```

Replacing Characters

- `erase`
  - One-argument version `(s.erase(i))`
    - Erases everything from (and including) the specified character position to the end of the string
- `replace`
  - Three-argument version `(s.replace( pos, n, str))`
    - Replaces characters at `pos` for `n` characters with the specified string `str`
  - Five-argument version `(s.replace( pos1, n1, str, pos2, n2))`
    - Replaces characters in the range starting from `pos1` for `n1` characters (specified by the first two arguments) with characters in the string `str` (third argument) from `pos2` for `n2` characters (specified by the last two arguments)

Some Other String Editing Operations

s.erase(pos, n)	Removes *n* characters from *s*, beginning at position *pos* ; returns *s*. With only one argument, it erases everything starting at position pos.
s.replace(pos1, n1, str)	Replaces the substring of *s* beginning at position *pos1* of length *n1* with string *str*; returns *s*
s.replace(pos1, n1, ca, n2)	Replaces a substring of *s* as before, but with the first *n2* characters in *ca*; returns *s*

string7.cpp and Sample Output

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

Original string after erase:
The values in any left subtree
are less than the value in the


After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the


After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

Inserting Characters

- insert
  - For inserting characters into a string
- Two-argument version `(s.insert( pos, str))`
  - First argument specifies insertion location (characters at and after `pos` in the original string will be pushed back)
  - Second argument specifies string to insert; the new string `str` will be inserted starting at position `pos`
- Four-argument version `(s.insert( pos, str, pos1, n))`
  - First argument specifies the starting insertion location
  - Second argument specifies string to insert from
  - Third and fourth arguments specify starting and number of elements in `str` to be inserted
  - Use `string::npos` for `n` causes the entire string from `pos1` to be inserted

String Editing Operations

s.insert(pos, str)	Inserts a copy of str into s at positon pos; returns s
s.insert(pos1, str, pos2, n)	Inserts a copy of *n* characters of *str*, starting at position *pos2*, into *s* at position *pos1*; if *n* is too large, characters are copied only until the end of *str* is reached; returns *s*
s.insert(pos, ca, n)	Inserts a copy of the first *n* characters of *ca* into *s* at position *pos*; inserts all of its characters if *n* is omitted; returns *s*
s.insert(pos, n, ch)	Inserts *n* copies of the character *ch* into *s* at position *pos*; returns *s*

string8.cpp and its Sample Output

```
Initial strings:
string1: beginning end
string2: middle
string3: 12345678
string4: xx

Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

String and C-style String Conversions

- When C-style string is needed instead of a string object
- Converts to an array of char
- `copy (i.e., s.copy( buffer ) )`
  - Copies current string into the specified char array
  - Must manually add terminating null character afterward
- `c_str (i.e., cptr = s.c_str() )`
  - Returns a `const char *` (rvalue) containing a copy of the current string which cannot be modified
  - Automatically adds terminating null character
  - If the original string object is later modified, this pointer may become invalid as the heap may be reallocated
  - Need to make copies of cptr array if you want to modify it by including `<cstring>`:

```
char * cstr = new char [str.size()+1];
strcpy (cstr, str.c_str());
```

- `data (i.e., cptr = s.data() )`
  - Returns *non*-null-terminated C-style character array
  - This pointer may become invalid if `s` is later changed
  - Printing out `cptr` may print out more than `s`, as it is not null-terminated

string9.cpp and its Sample Output

- C-style string conversion
- Sample Output

```
string string1 is STRINGS
string1 converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

Some Common string Member Functions

- string::append
- string::assign
- string::at
- string::begin
- string::capacity
- string::clear
- string::compare
- string::copy
- string::c_str
- string::data
- string::empty
- string::end
- string::erase
- string::find
- string::find_first_not_of
- string::find_first_of
- string::find_last_not_of
- string::find_last_of
- string::insert
- string::length
- string::max_size
- string::operator+=
- string::operator=
- string::operator[]
- string::push_back
- string::rbegin
- string::rend
- string::replace
- string::resize
- string::rfind
- string::size
- string::substr
- string::swap