Advanced Computer Contest Preparation

Lecture 1

REVIEW

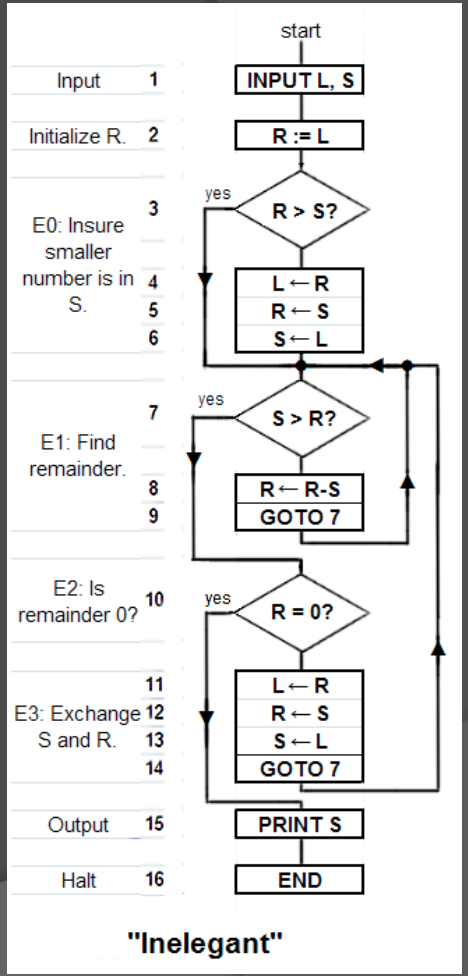
Contents

- Review of Grade 12 concepts necessary for this course:
  - Algorithms
  - Recursion
  - Pointers and Iterators
  - Sorting
  - Binary Search
  - Basic/Built-in Data Structures
  - Documentation

Algorithms

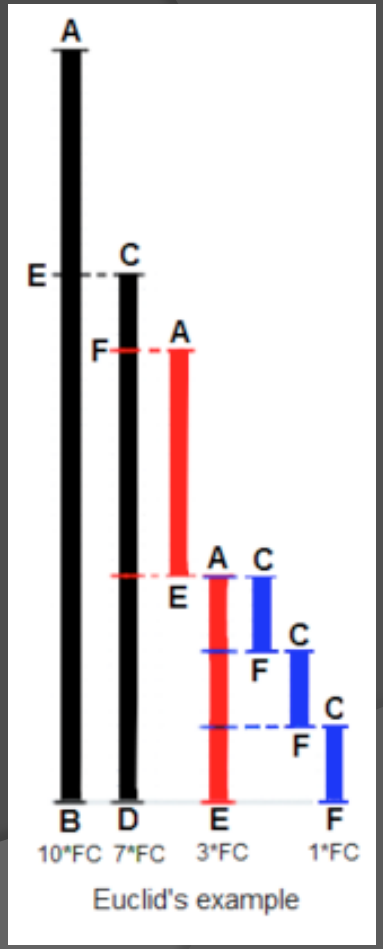
Algorithms

- A process to be followed in calculations/problem-solving operations (Google)
- A step-by-step set of operations to be performed (Wikipedia)
- A specific procedure that solves a certain problem
  - A solution to a contest problem



Examples of Algorithms

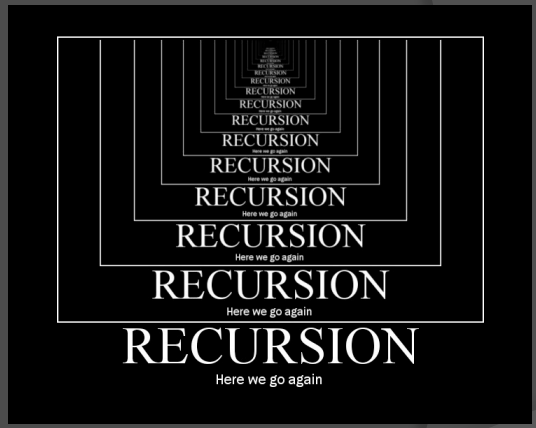
- Adding two numbers
- Finding the largest value in a randomly ordered list
- Euclid's Algorithm
  - Finds the greatest common denominator (GCD) of two numbers



Recursion

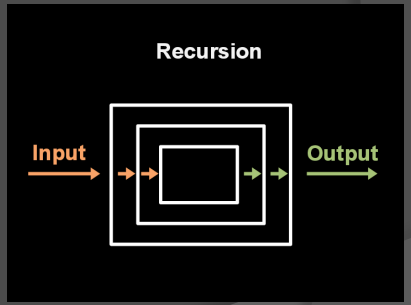
Recursion

- A problem solving technique
- Solution of a problem depends on smaller instances of the problem (subproblem)
- Solve the smaller problems and combine their answers to solve larger problems
  - Referred to as the **divide and conquer method**
- **Recursive functions** are required



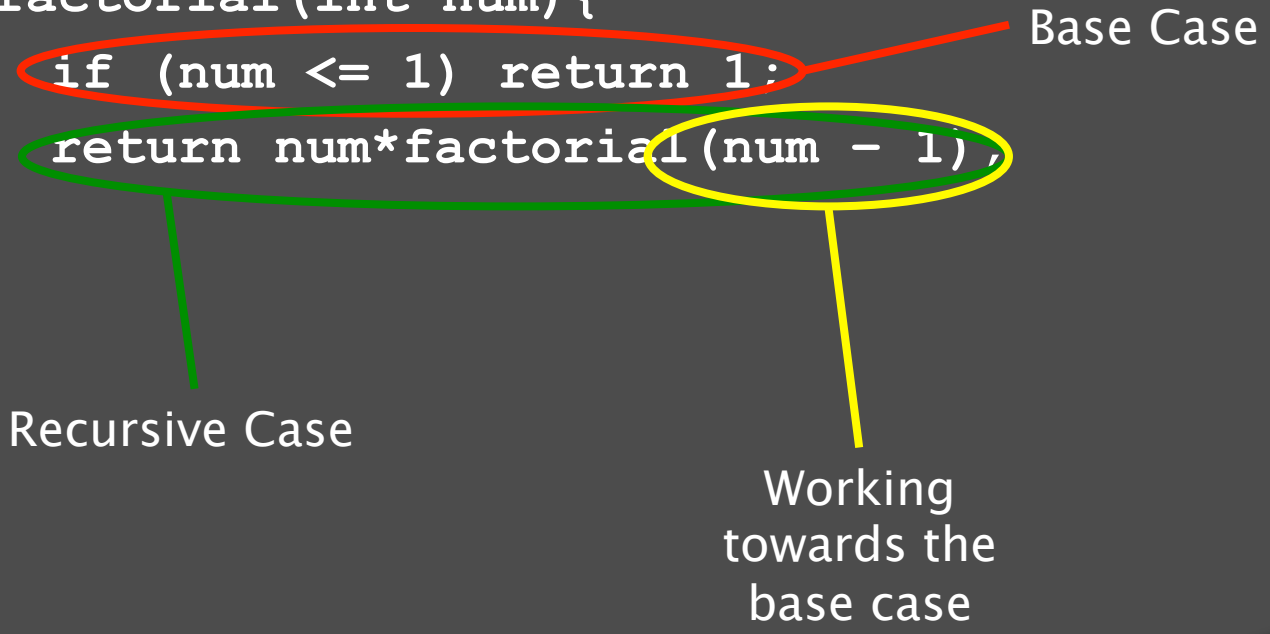
Recursive Functions

- A function that calls itself somewhere in its body
- Rules of Recursion:
  - Must have a **base case**
    - Point where problem subdividing ends and a simple solution can be generated
  - Must have a **recursive case**
    - A recursive method must call itself
  - Must **work towards** base case
  - **Do not repeat work!**
    - Avoid re-computing solutions to problems already encountered



Recursive Function Example

```
int factorial(int num){
    if (num <= 1) return 1;
    return num*factorial(num - 1);
}
```



Recursive Function Example

```
int factorial(int num){
        if (num <= 1) return 1;
        return num*factorial(num - 1);
}
```

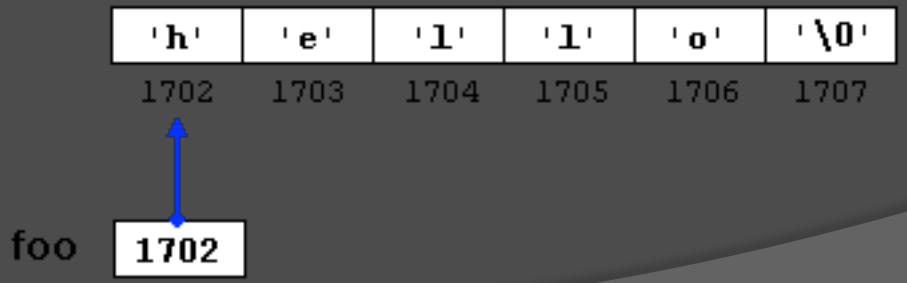
Trace:

factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
factorial(2) = 2 * 1 = 2
factorial(3) = 3 * 2 = 6
factorial(4) = 4 * 6 = 24
factorial(5) = 5 * 24 = 120

Pointers and Iterators

Pointers

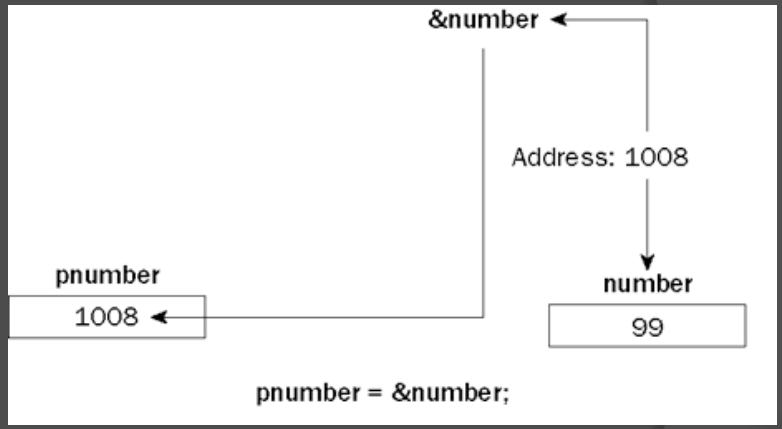
- Variables and objects are all stored at specific locations in memory
  - Can be accessed by their identifier (name)
- Pointers are variables that store memory addresses
  - Can be used to access variables



Address-of operator (&)

- Get the memory address of a variable using the **address-of operator**

```
foo = &myvar
```



- **`foo`** is a pointer and **`myvar`** is a variable
- Note: the memory address of a variable cannot be determined before running the program (and it can be different on different runs)

Dereference operator (*)

- Pointers are said to “point to” the variable whose address they store

- Pointers can be used to access the variable they point to directly by using the **dereference operator**

```
myothervar = *foo //myothervar == myvar
(*foo) += 5 //adds 5 to myvar
```

Declaring Pointers

- Declaring a pointer follows this syntax:

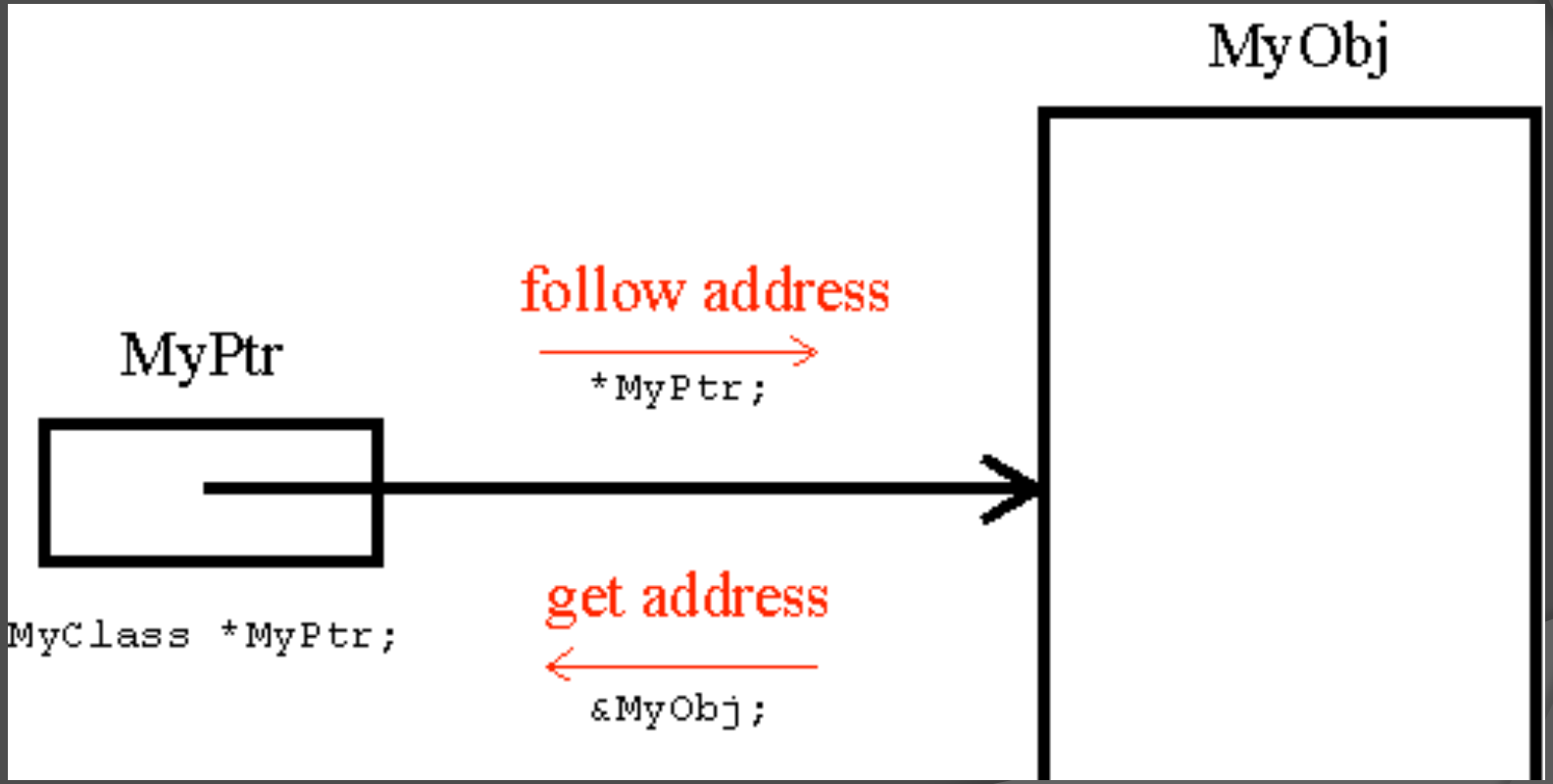
```
type * name
```

- Note that the asterisk (*) is both used as the dereference operator and to declare a pointer
- `type` is the type of variable that the pointer is pointing to.
  - For instance, `int * ptr` must point to an `int`

Pointer Arithmetic

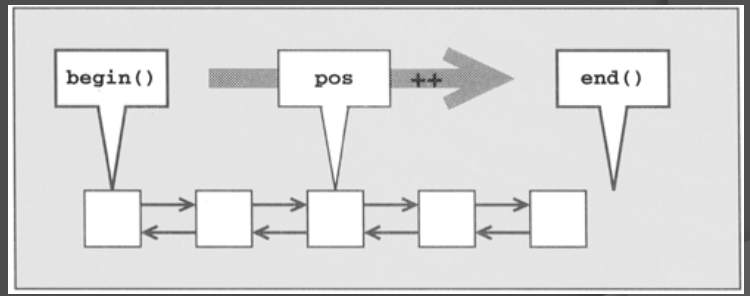
- Adding and subtracting from a pointer will change the memory address of what the pointer is pointing to
- Typically not done in contest programming unless iterators are being operated on
- To change the variable the pointer is pointing to, remember to use the dereference operator (*)

Pointer Summary



Iterators

- An object that points to some element in a range of elements (e.g. array or container)
- An iterator can **iterate** through the elements of that range using operators (at least increment (++) and dereference (*))
- **Not all iterators have the same functionality of pointers**
  - e.g. `it++` might give the next element of a map, but its memory address is not the next available one



Iterator Example

Get the sum of all elements in a set:

```
int sum = 0;
for (set<int>::iterator it = nums.begin(); it != nums.end(); it++){
        sum += *it;
}
printf("%d\n",sum);
```

RandomAccessIterator

- Random-access iterators are iterators that can be used to access elements at an arbitrary offset position relative to the element they point to, offering the same functionality as pointers.
- Arrays and some other data structures are accessed using RandomAccessIterators

RandomAccessIterator - Offset

- The name of the array is a pointer to its first element
  - For example, **`arr`** points to the first element in **`arr[]`**
- Objects that use RandomAccessIterators normally store elements in **consecutive blocks in memory**
- Thus, to access the n^{th} element in **`arr[]`** using pointers, add **`n-1`** to **`arr`**
  - e.g. **`arr+5`** is the location of the 6^{th} element in **`arr[]`**
- Can be done on any RandomAccessIterator

RandomAccessIterator - Offset

- Since adding to an RandomAccessIterator is possible, we can find out the number of elements in between two given RandomAccessIterators by subtracting
  - e.g. `arr+7 - arr` will give a result of 7, meaning that there are 7 elements between the element 0, inclusive, and element 7, exclusive
- Mostly useful when used with binary search methods

RandomAccessIterator

- Arrays are not the only users of RandomAccessIterators!
- **`string`**, **`vector`**, and other objects use RandomAccessIterators as well
- Objects have methods such as **`begin()`** and **`end()`** to access these iterators

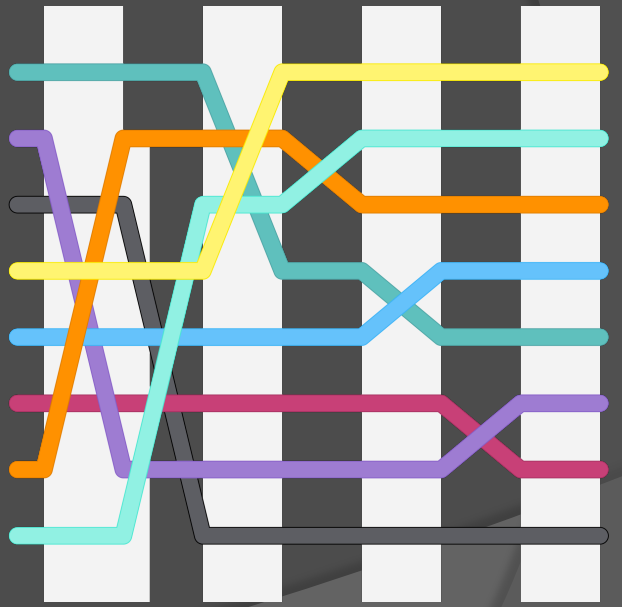
Iterators that are not RandomAccessIterators

- Not all iterators are RandomAccessIterators!
  - For example, **`map`** and **`set`** do not use them
- The difference is that **offset** cannot be used if the iterator is not a RandomAccessIterator
  - e.g. `it+5` will not give the sixth element
- Exception is `it++` and `it--`, they should point to the next element

Sorting

Sorting

- Sorting is a frequently used operation in contests (and in general programming)
- You **DO NOT** need to implement your own sorting algorithm in most cases
  - It may be good to know some fast sorting algorithms just in case you need to implement your own
- Use built-in sorting methods whenever possible



Sorting Algorithms

- Generally Slow:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort (best)
- Generally Fast:
  - Heap Sort
  - Merge Sort
  - Quick Sort (best)

The standard sorting algorithm in C++ uses a combination of these 3 sorts!

C++ Sorting

*Don't forget "#include <algorithm>" at the top

This method sorts all elements in the range [first,last): first, inclusive, to last, exclusive.

Default:

```
void sort (RandomAccessIterator first,
RandomAccessIterator last);
```

```
int ar[5] = {5,4,3,2,1};
sort(ar,ar+5); //ar becomes {1,2,3,4,5}
```

C++ Custom Sorting

- C++ allows you to specify the compare function to use to sort

```
void sort (RandomAccessIterator first,
RandomAccessIterator last, Compare comp);

bool cmp(int a, int b){return a > b;}
int ar[5] = {4,2,5,1,3};
sort(ar,ar+5,cmp); //ar becomes {5,4,3,2,1}
```

Binary Search

Searching

- How can we see if a certain number exists in this array:

```
1,4,6,9,11,34,69,99,201,334,420,1002,9001
```

- Is there a better method than searching linearly from the first element to the last?

Binary Search

- Algorithm:
  - Check the middle element of the search area
  - If the target value is less, then look at the left
  - If the target value is greater, then look at the right
  - Otherwise, the target value has been found!
- The search area is halved each time, making binary search efficient
- Note that the area **must be sorted**

Binary Search C++ Code

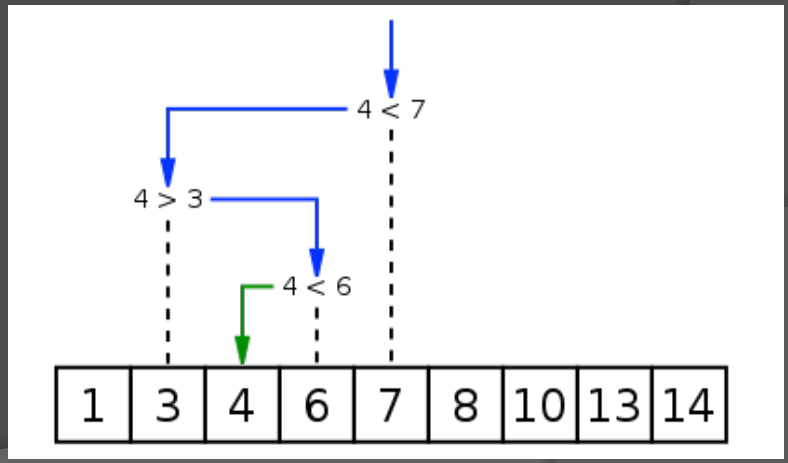
```
int search(int ar[], int target, int size){
      int lo = 0, hi = size-1;
      while(lo < hi){
            int mid = (lo+hi)/2;
            if (ar[mid] == target) return mid;
            else if (ar[mid] < target) lo = mid+1;
            else hi = mid-1;
      }
      //target was not found
      return -1
}
```

Binary Search

- Like sorting, binary searching is built-in in various languages, including C++
- There are 3 important C++ binary search methods:
  - binary_search()
  - lower_bound()
  - upper_bound()
- Using both upper_bound() and lower_bound() will give you a range where all elements are equal to the given value

Binary Search

- Unlike sorting, there are more problems where custom implementation of a binary search are required.
- Try to use built-in methods whenever possible



Binary Search Methods

*Don't forget "#include <algorithm>" at the top!

```
bool binary_search (ForwardIterator first,
ForwardIterator last, const T& val);
```

Returns true if an element in [first,last) is equivalent to val

```
int ar[10] = {1,2,3,4,5,6,7,8,9,10};
cout << binary_search(ar,ar+10,6) << endl;  //true
cout << binary_search(ar,ar+10,20) << endl; //false
```

Binary Search Methods

```
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator
last, const T& val);
```

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than val (element >= val)

```
int ar[10] = {2,2,4,4,6,6,8,8,10,10};
cout << lower_bound(ar,ar+10,6) - ar; //4
```

Binary Search Methods

```
ForwardIterator upper_bound
(ForwardIterator first, ForwardIterator
last, const T& val);
```

Returns an iterator pointing to the first element in the range [first,last) which compares greater than val (element > val)

```
int ar[10] = {2,2,4,4,6,6,8,8,10,10};
cout << upper_bound(ar,ar+10,6) - ar; //6
```

Data Structures

Containers

- A holder object that stores a collection of other objects (its elements)
- Container manages storage space for its elements and provides member functions to access them
- Typically implemented as **class templates** to allow flexibility in types supported as elements

Class Template

- Classes in the form of:

```
class_name <object_type,...> obj_name;
```

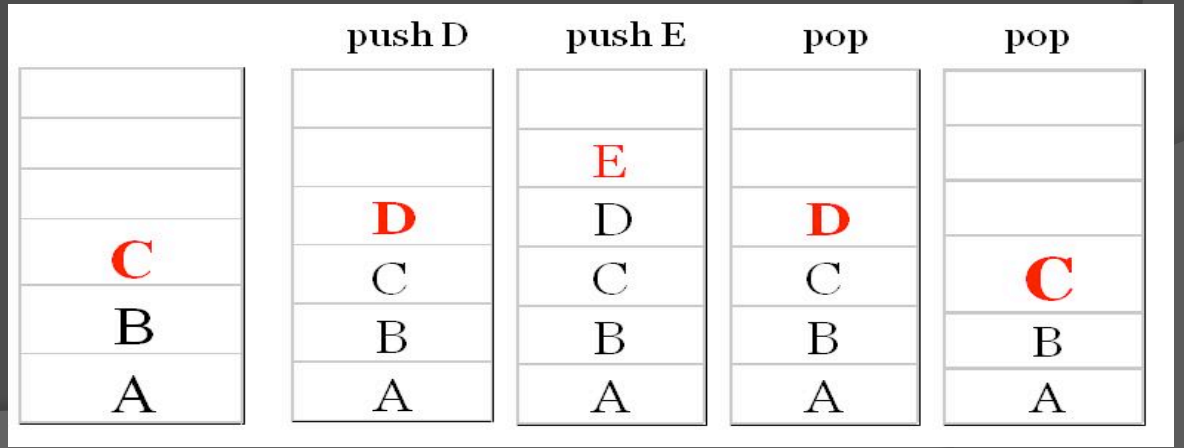
- Vectors, stacks, and queues all use this format
- Examples:
  - `vector<int> vect;`
  - `stack<AnObject> st;`
  - `queue<string> q;`
  - `map<int,int> mp;`

Vector

- Arrays that can change in size
- Also called **dynamic arrays**
- In C++, you can use [] to get and modify elements of vector
- Efficient at accessing elements and adding/ removing from end
- Use C++ documentation to find useful vector methods

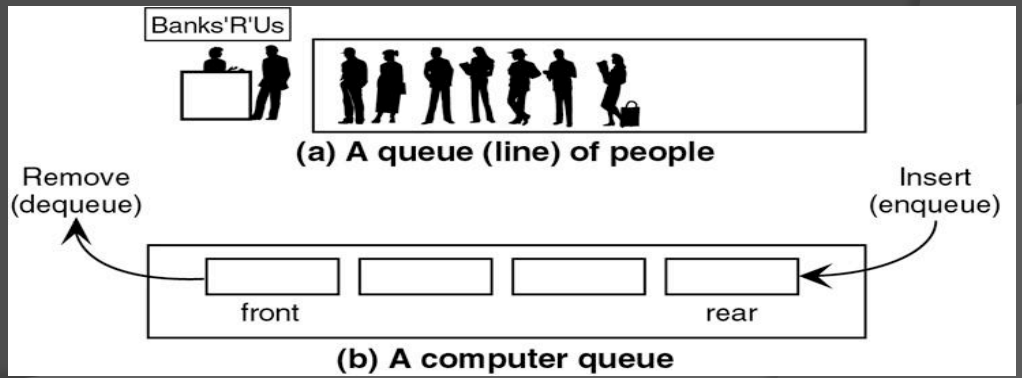
Stack

- A LIFO (last-in first-out) container
- Functions like a stack of papers
  - Cannot access papers other than the top without moving other papers



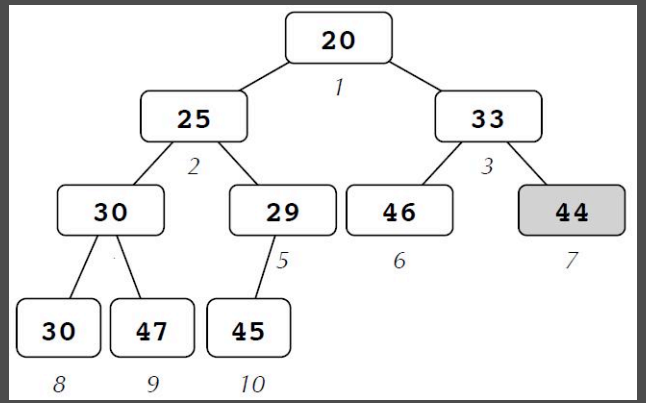
Queue

- A FIFO (first-in first-out) container
- Functions like a queue of people
  - People who arrive first are serviced first
  - “First come, first serve”



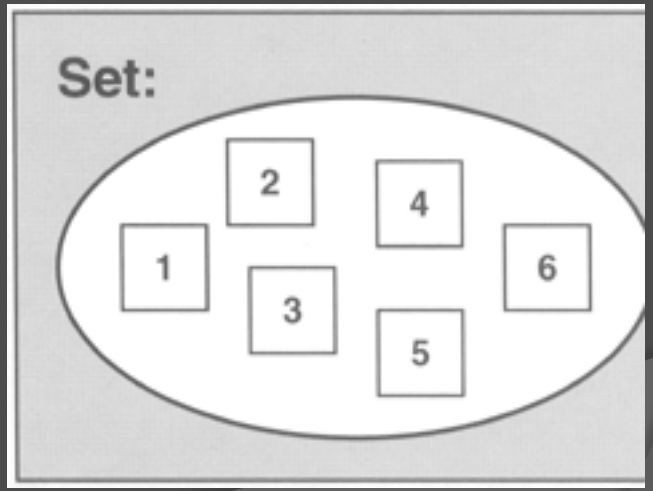
Priority Queue

- A container where the first element is always the greatest or least of all of its elements
- Functions similar to priority lines at airports
- Compare function can be custom



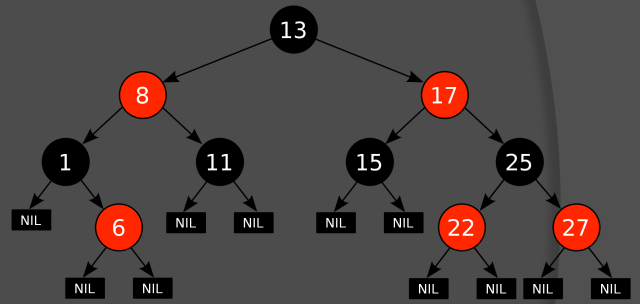
Set

- Containers that store unique elements.
- Two types of sets:
  - `set`
  - `unordered_set`



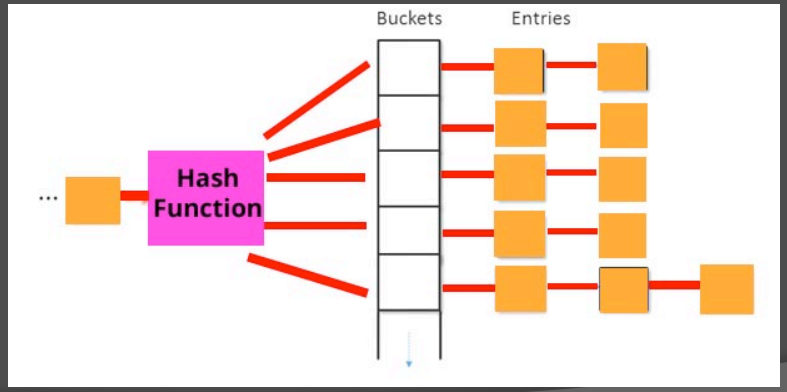
Set

- Elements are stored following a specific order
- Ordering criterion can be custom
- Typically slower than **`unordered_set`**, but still fast
- Implemented as balanced binary search trees
- Elements can be iterated in order



Unordered_set

- Stores elements in no particular order
- Elements are hashed for quick lookup
- Faster to access elements than **`set`**
- Elements are not iterated in order



Map

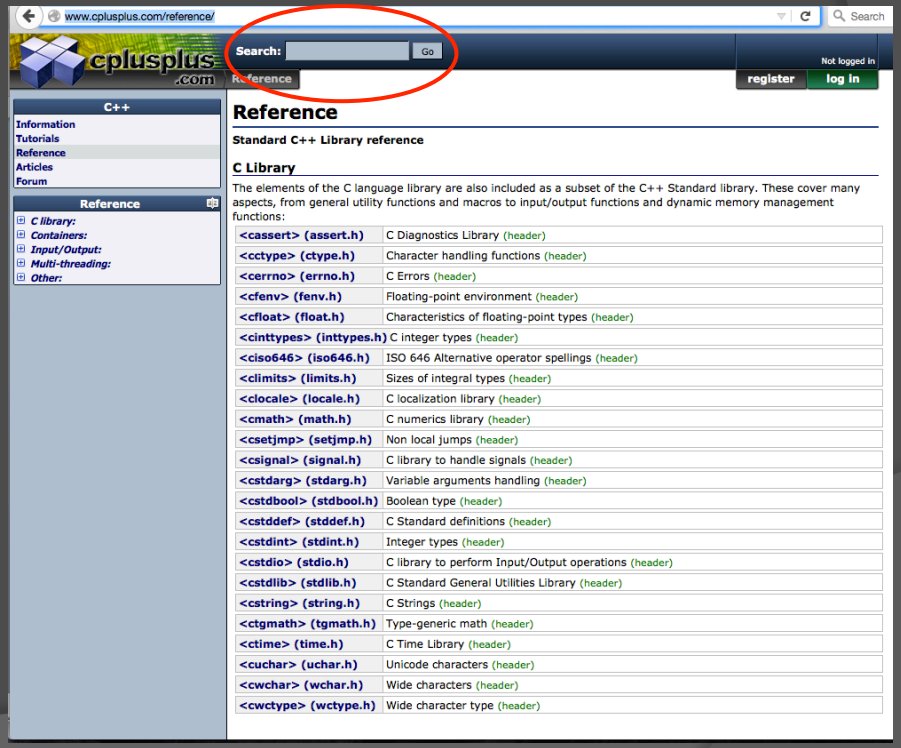
- Associative containers that store elements formed by a combination of a **key value** and a **mapped value**
- Key values are used to identify elements
- [] operator can be used to access elements
- Two types of maps:
  - `map`
  - `unordered_map`
- Like `set`, the type of map how keys are stored (sorted or hashed, respectively)

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

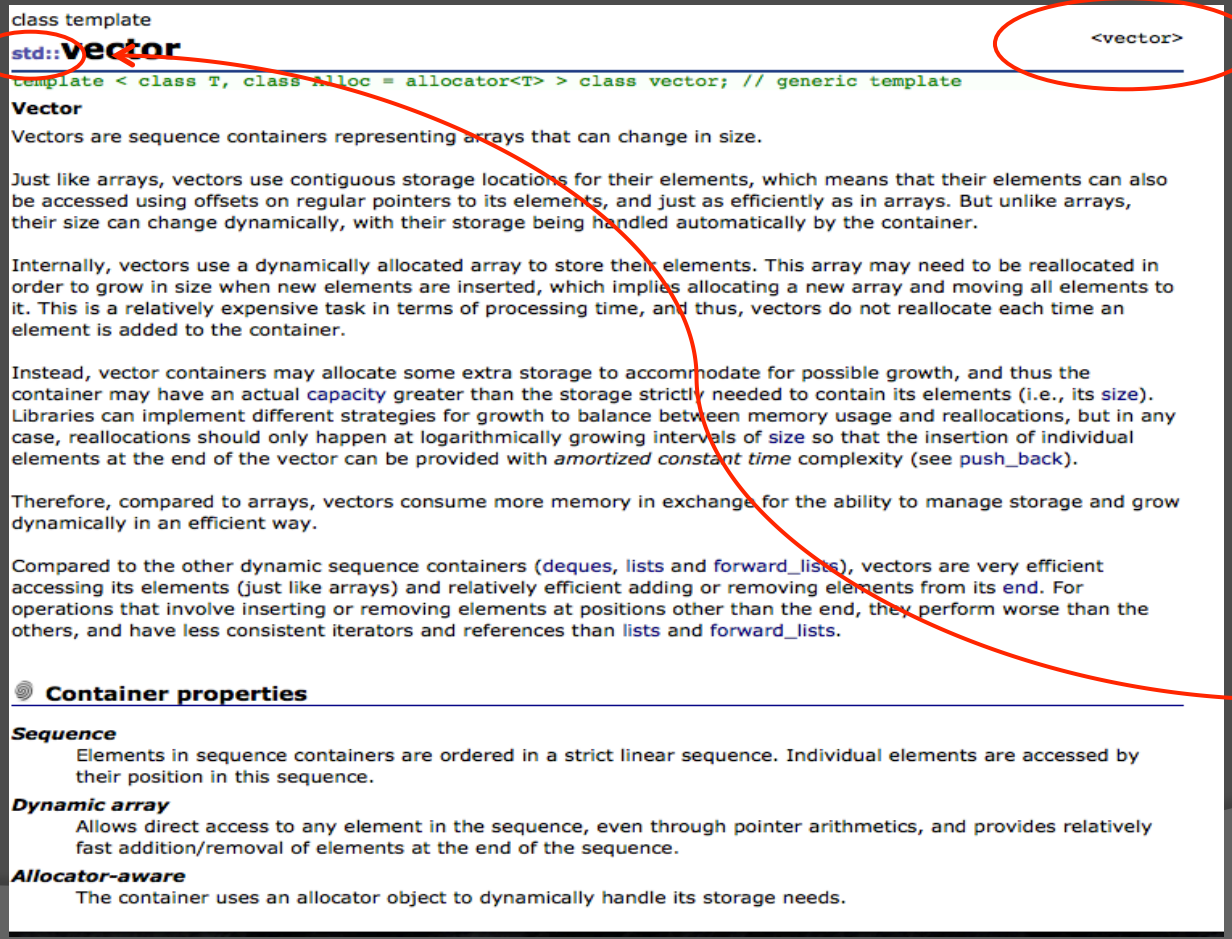
Documentation

Documentation

- http://www.cplusplus.com/reference/
- USE THE SEARCH BAR



Library Imports



class template

std::vector

<vector>

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

Vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

Container properties

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Dynamic array
Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

Allocator-aware
The container uses an allocator object to dynamically handle its storage needs.

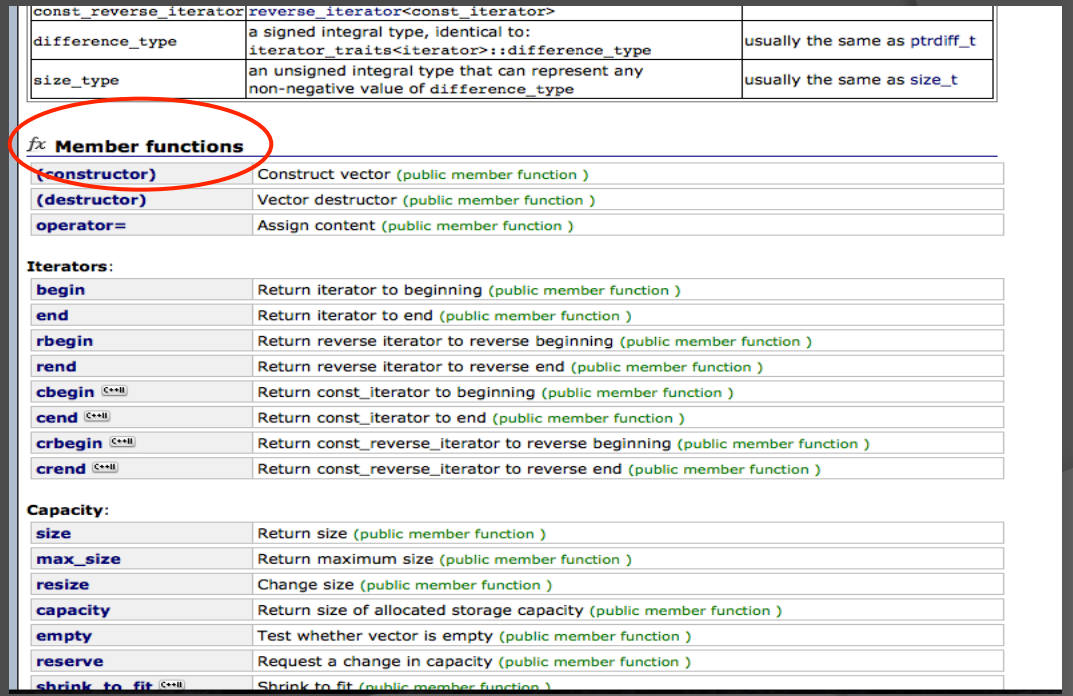
THIS LIBRARY MUST BE INCLUDED AT THE TOP OF YOUR CODE

#include <library_name>

Namespace required

Functions of a Built-in Class

It's all here!



const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	a signed integral type, identical to: iterator_traits<iterator>::difference_type	usually the same as ptrdiff_t
size_type	an unsigned integral type that can represent any non-negative value of difference_type	usually the same as size_t

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

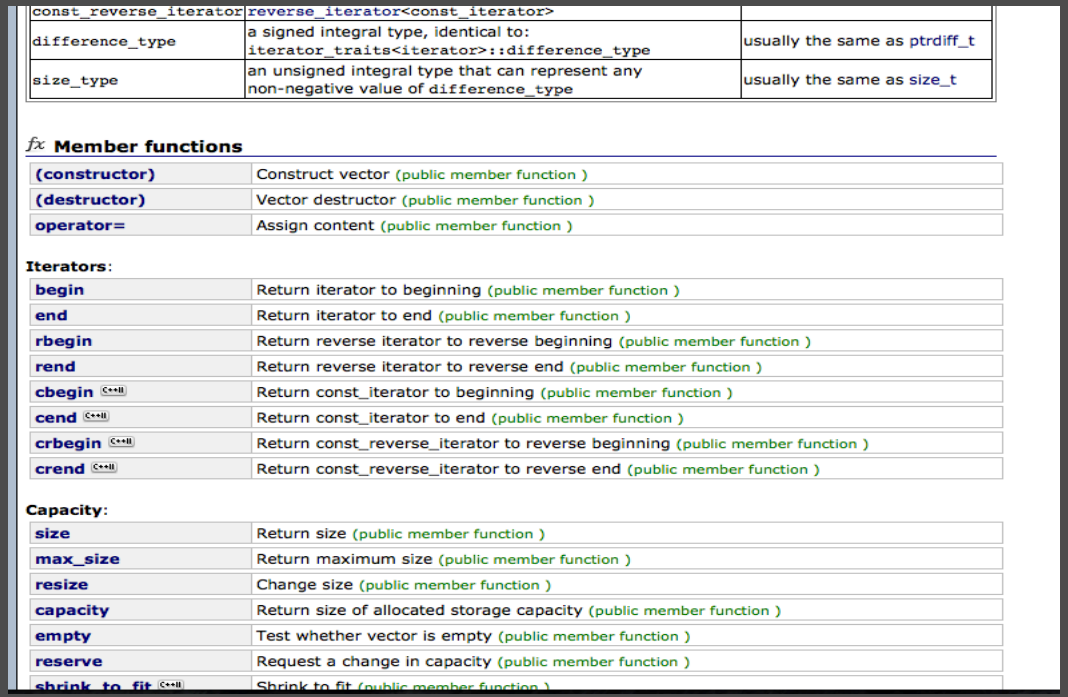
Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Functions of a Built-in Class



const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	a signed integral type, identical to: iterator_traits<iterator>::difference_type	usually the same as ptrdiff_t
size_type	an unsigned integral type that can represent any non-negative value of difference_type	usually the same as size_t

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

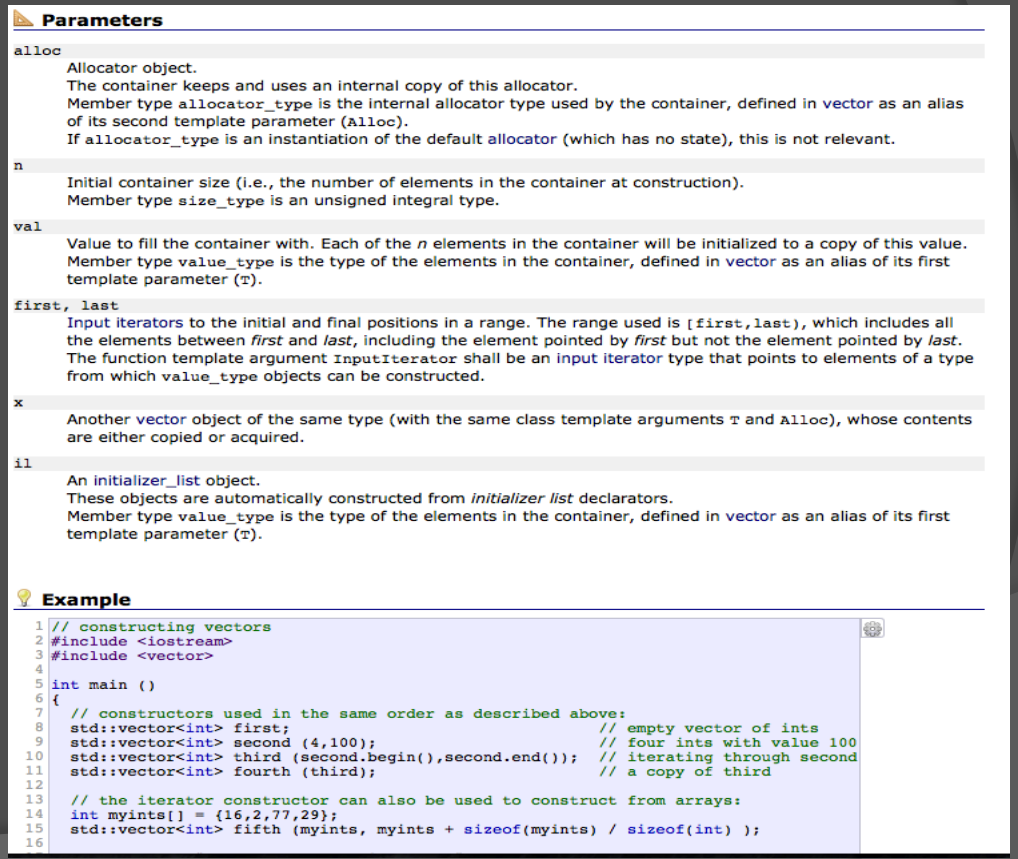
Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Click on any function name...

Functions of a Built-in Class

Get information such as parameters, sample code, and time complexity!



Parameters

`alloc`
Allocator object.
The container keeps and uses an internal copy of this allocator.
Member type `allocator_type` is the internal allocator type used by the container, defined in vector as an alias of its second template parameter (`Alloc`).
If `allocator_type` is an instantiation of the default allocator (which has no state), this is not relevant.

`n`
Initial container size (i.e., the number of elements in the container at construction).
Member type `size_type` is an unsigned integral type.

`val`
Value to fill the container with. Each of the *n* elements in the container will be initialized to a copy of this value.
Member type `value_type` is the type of the elements in the container, defined in vector as an alias of its first template parameter (`T`).

`first, last`
Input iterators to the initial and final positions in a range. The range used is `[first,last)`, which includes all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.
The function template argument `InputIterator` shall be an input iterator type that points to elements of a type from which `value_type` objects can be constructed.

`x`
Another vector object of the same type (with the same class template arguments `T` and `Alloc`), whose contents are either copied or acquired.

`il`
An initializer_list object.
These objects are automatically constructed from *initializer list* declarators.
Member type `value_type` is the type of the elements in the container, defined in vector as an alias of its first template parameter (`T`).

Example

```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  // constructors used in the same order as described above:
  std::vector<int> first;                                // empty vector of ints
  std::vector<int> second (4,100);                       // four ints with value 100
  std::vector<int> third (second.begin(),second.end());  // iterating through second
  std::vector<int> fourth (third);                       // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
```

THANK YOU!