# Standard Template Library (STL)

# The C++ STL

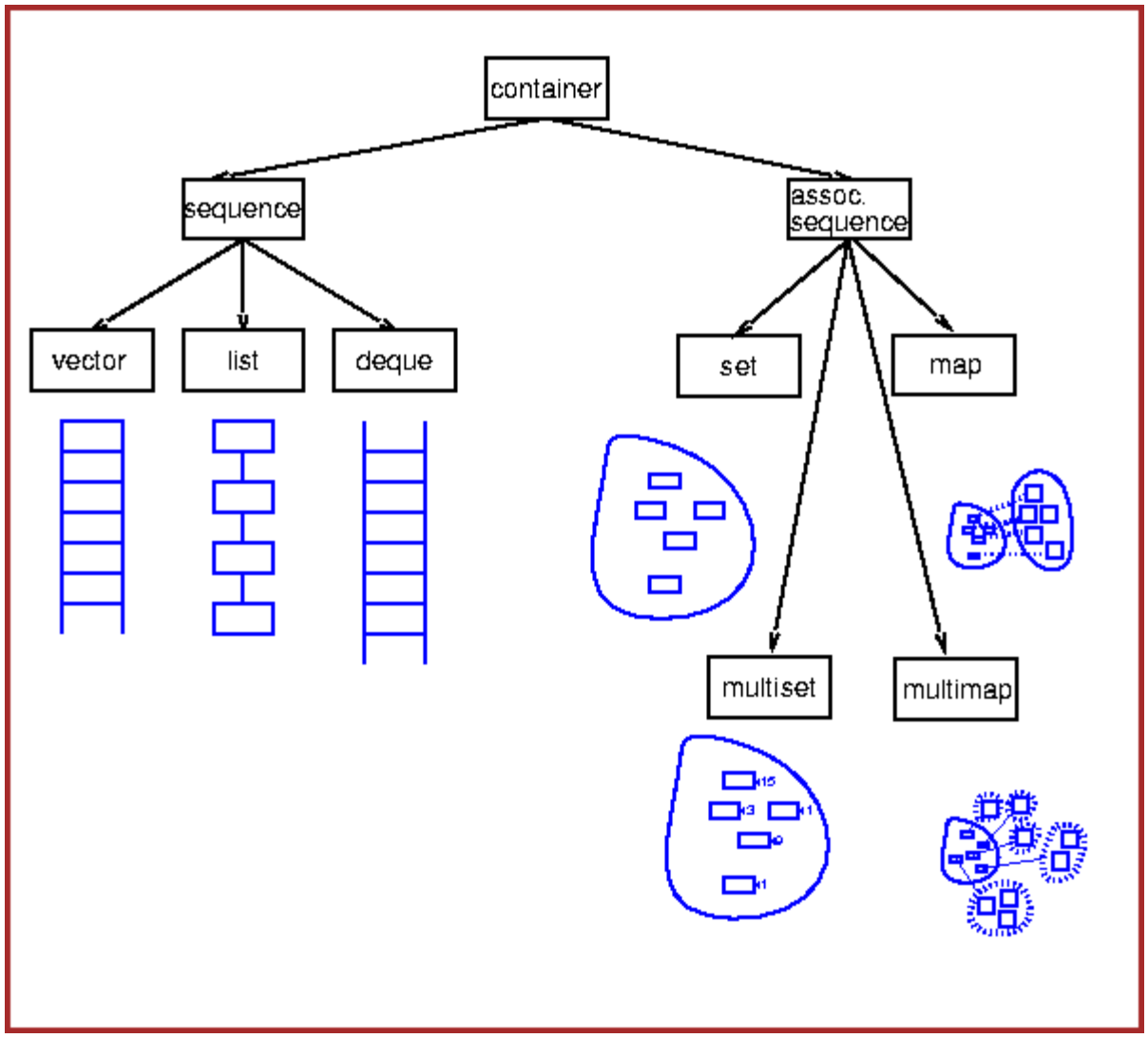
- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.

# Components of the STL

- Program's main objective is to manipulate data and generate results
  - Requires ability to **store** data, **access** data, and **manipulate** data
- STL has three basic components:
  **(1)** **Containers**: generic class templates for **storing** collection of data (contain other objects).
  **(2)** **Iterators:** generalized 'smart' **pointers** that provides operations for indirect access and facilitate use of containers. They provide an interface that is needed for STL algorithms to operate on STL containers.
  **(3)** **Algorithms**: generic **function templates** for operating on containers.

# Why use STL?

- STL offers an assortment of **containers**
- STL publicizes the time and storage **complexity** of its containers
- STL containers grow and shrink in **size** automatically
- STL provides built-in **algorithms** for processing containers
- STL provides **iterators** that make the containers and algorithms flexible and efficient.
- STL is **extendable** which means that users can add new containers and new algorithms.
- **Memory management**: no memory leaks or serious memory-access violations. (e.g., pointers)
- Reduce testing and debugging **time**.


container
sequence
assoc. sequence
vector
list
deque
set
map
multiset
multimap

# Sequence Containers

- Every object has a specific position
- Predefined sequence containers
  - `vector`, `deque`, `list`
- Sequence container `vector`
  - Logically: same as **arrays**
- All containers
  - Use same names for common operations
  - Have specific operations

# Sequence Container: `vector`

- Vector container
  - Stores, manages objects in a **dynamic array**
  - Elements accessed **randomly**
  - Time-consuming item insertion: beginning and middle
  - Fast item insertion: end
- Class implementing vector container
  - `vector`
- Header file containing the `class vector`
  - `vector`
- Using a vector container in a program requires the following statement:
  - `#include <vector>`

- Declaring vector objects

Various ways to declare and initialize a vector container

| Statement | Effect |
|---|---|
| `vector<elementType> vecList;` | Creates an empty vector, `vecList`, without any elements. (The default constructor is invoked.) |
| `vector<elementType> vecList(otherVecList);` | Creates a vector, `vecList`, and initializes `vecList` to the elements of the vector `otherVecList`. `vecList` and `otherVecList` are of the same type. |
| `vector<elementType> vecList(size);` | Creates a vector, `vecList`, of size `size`. `vecList` is initialized using the default constructor. |
| `vector<elementType> vecList(n, elem);` | Creates a vector, `vecList`, of size `n`. `vecList` is initialized using `n` copies of the element `elem`. |
| `vector<elementType> vecList(begin, end);` | Creates a vector, `vecList`. `vecList` is initialized to the elements in the range `[begin, end)`, that is, all elements in the range `begin...end-1`. |

– Examples:
  - `vector<int> intlist;`
  - `vector<string> stringList;`

# Operations to **access** the elements of a vector container

| Expression | Effect |
|---|---|
| `vecList.at(index)` | Returns the element at the position specified by `index`. |
| `vecList[index]` | Returns the element at the position specified by `index`. |
| `vecList.front()` | Returns the first element. (Does not check whether the container is empty.) |
| `vecList.back()` | Returns the last element. (Does not check whether the container is empty.) |

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

int main()
{
std::vector<int> myvector(10); // 10 zero-initialized ints

  // assign some values:
for (unsigned i = 0; i<myvector.size(); i++)
myvector.at(i) = i;

std::cout << "myvector contains:";
for (unsigned i = 0; i<myvector.size(); i++)
std::cout << ' ' << myvector.at(i);
std::cout << '\n';

return 0;}
```

```
myvector contains: 0 1 2 3 4 5 6 7 8 9
```

# Declaring an Iterator to a Vector Container

- Process vector container like an array
  - Using array subscripting operator
- Process vector container elements
  - Using an iterator
- `class vector`: function `insert`
  - Insert element at a specific vector container position
  - Uses an iterator
- `class vector`: function `erase`
  - Remove element
    - Uses an iterator

- `class vector` contains `typedef iterator`
  - Declared as a public member
  - Vector container iterator
  - Example

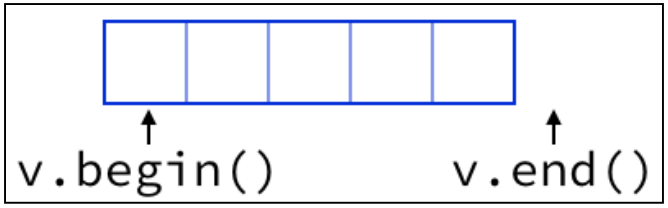
```
vector<int>::iterator intVecIter;
```

- Requirements for using `typedef iterator`
  1. Container name (vector)
  2. Container element type (<int>)
  3. Scope resolution operator (::)

- **`++intVecIter`**
  - Advances iterator `intVecIter` to next element into the container
- **`*intVecIter`**
  - Dereferencing
  - Returns element at current iterator position

# Containers and the Functions `begin` and `end`

- A sequence is defined by a pair of iterators defining a half-open range **[begin:end)**
  - Includes first element but excludes last element.
- **`begin`**
  - Returns an iterator to the first element in the container
- **`end`**
  - Returns an iterator to the element past the end. It does not point to any element. Never read from or write to `*`end.



```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{ vector<int> v1;
v1.push_back(2);
v1.push_back(4);
v1.push_back(7);
vector<int> v2(v1);
vector<int> v3(3);
v3.at(0) = 4;
v3.at(1) = 6;
v3.at(2) = 4;
vector<int> v4(4, 2);
vector<int> v5(v2.begin(), v2.end());

for (unsigned i = 0; i < v1.size(); i++)
{cout << ' ' << v1.at(i) << "\t" << v2[i] << "\t" << v3.at(i) << "\t" <<
v4.at(i) << "\t"<< v5.at(i);
cout << '\n';}

return 0;}
```

| | | | | |
|---|---|---|---|---|
| 2 | 2 | 4 | 2 | 2 |
| 4 | 4 | 6 | 2 | 4 |
| 7 | 7 | 4 | 2 | 7 |

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<int> v1;
v1.push_back(3);
v1.push_back(4);
v1.push_back(6);
vector<int>::iterator it;

cout << v1.front() << v1.back() << "\n";

for (it = v1.begin(); it != v1.end(); it++)
cout << *it;

return 0;}
```

```
36
346
```

# Various operations on a vector container

| Expression | Effect |
|---|---|
| `vecList.clear()` | Deletes all elements from the container. |
| `vecList.erase(position)` | Deletes the element at the position specified by position. |
| `vecList.erase(beg, end)` | Deletes all elements starting at `beg` until `end-1`. |
| `vecList.insert(position, elem)` | A copy of `elem` is inserted at the position specified by `position`. The position of the new element is returned. |
| `vecList.insert(position, n, elem)` | `n` copies of `elem` are inserted at the position specified by `position`. |
| `vecList.insert(position, beg, end)` | A copy of the elements, starting at `beg` until `end-1`, is inserted into `vecList` at the position specified by `position`. |

- **position is an iterator**
- **insert():** the vector is extended by inserting new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.
- **Return value:** an **iterator** that points to the first of the newly inserted elements.

| Expression | Effect |
| --- | --- |
| `vecList.push_back(elem)` | A copy of `elem` is inserted into `vecList` at the end. |
| `vecList.pop_back()` | Deletes the last element. |
| `vecList.resize(num)` | Changes the number of elements to `num`. If `size()`, that is, the number of elements in the container increases, the default constructor creates the new elements. |
| `vecList.resize(num, elem)` | Changes the number of elements to `num`. If `size()` increases, the default constructor creates the new elements. |

```cpp
// erasing from vector
#include <iostream>
#include <vector>
int main()
{
std::vector<int> myvector;
// set some values (from 1 to 10)
for (int i = 1; i <= 10; i++) myvector.push_back(i);

// erase the 7th element
myvector.erase(myvector.begin() + 6);

// erase the first 3 elements:
myvector.erase(myvector.begin(), myvector.begin() + 3);

std::cout << "myvector contains:";
for (unsigned i = 0; i<myvector.size(); ++i)
 std::cout << ' ' << myvector[i];
std::cout << '\n';
return 0;
}
```

```
myvector contains: 4 5 6 8 9 10
```

```cpp
#include <iostream>
#include <vector>
int main(){
std::vector<int> myvector(3, 100);
std::vector<int>::iterator it;
it = myvector.begin();
it = myvector.insert(it, 200);
myvector.insert(it, 2, 300);
// "it" no longer valid, get a new one:
it = myvector.begin();

std::vector<int> anothervector(2, 400);
myvector.insert(it + 2, anothervector.begin(), anothervector.end());

int myarray[] = { 501,502,503 };
myvector.insert(myvector.begin(), myarray, myarray + 3);

std::cout << "myvector contains:";
for (it = myvector.begin(); it<myvector.end(); it++)
std::cout << ' ' << *it; return 0;}
```

myvector contains: 501 502 503 300 300 400 400 200 100 100 100

```cpp
#include <iostream>
#include <vector>
int main()
{
std::vector<int> myvector;
// set some initial content:
for (int i = 1; i<10; i++) myvector.push_back(i);
myvector.resize(5);
myvector.resize(8, 100);
myvector.resize(12);

std::cout << "myvector contains:";
for (int i = 0; i<myvector.size(); i++)
std::cout << ' ' << myvector[i];
std::cout << '\n';

return 0;}
```

```
myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

# The `sort` Algorithm

- Sorts the elements in the range [first,last) into ascending order.
- `void sort (Iterator first, Iterator last);`
- `#include <algorithm>`

Write a program that can read any number of integers from the user, stores them in a vector, sorts them, and print the result.

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int main() {
int input;
vector<int> ivec;

// input
while (cin >> input )
ivec.push_back(input);

sort(ivec.begin(), ivec.end());

vector<int>::iterator it;

for ( it = ivec.begin(); it != ivec.end(); ++it )
 cout << *it << " ";

return 0;
}
```

# Generate random number

- **int rand (void);**
  - Returns a pseudo-random integral number in the range between 0 and RAND_MAX, which is a constant defined in <cstdlib>.
  - This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called.
  - This algorithm uses a **seed** to generate the series, which should be initialized to some distinctive value using function **srand**.
  - Notice though that this modulo operation does not generate **uniformly distributed random numbers** in the span
- A typical way to generate trivial pseudo-random numbers in a determined range using rand is to use the modulo of the returned value by the range span and add the initial value of the range:
  - v1 = rand() % 100;  // v1 in the range 0 to 99
  - v2 = rand() % 100 + 1;  // v2 in the range 1 to 100

- void **srand** (unsigned int seed);
  - **Initialize** random number generator
  - The pseudo-random number generator is initialized using the argument passed as seed.
  - For every different seed value used in a call to srand, the pseudo-random number generator can be expected to **generate a different succession of results in the subsequent calls to rand**.
  - Two different initializations with the same seed will generate the same succession of results in subsequent calls to rand.
  - If seed is set to **1**, the generator is reinitialized to its **initial value** and produces the same values as before any call to rand or srand.
  - In order to generate random-like numbers, srand is usually initialized to some distinctive runtime value, like the value returned by function **time** (declared in header <ctime>). This is distinctive enough for most trivial randomization needs.

```
#include <iostream>
#include <cstdlib>      /* srand, rand */
#include <ctime>        /* time */
using namespace std;
int main()
{
cout << "First number: " << rand() << endl;

srand(time(NULL));
for (int i = 0; i <5; i++)
cout << "Random number: " << rand() << endl;

srand(1);
cout << "Again the first number: " << rand();
getchar();
return 0;
}
```

# Passing arguments by reference

- When passing arguments by value, the only way to return a value back to the caller is via the function's **return** value.
- One way to allow functions to modify the value of argument is by using **pass by reference.**

```
void AddOne(int &y) // y is a reference variable
{y = y + 1;}
```

- When the function is called, y will become a reference to the argument. **Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument.**
- More: http://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/

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

void passByReference(int &y) // y is a reference
{   y = 7;}

void passByValue(int y) // y is a copy
{   y = 6;}

int main()
{
  int x = 5;
  passByValue(x);
  cout << "x = " << x << endl;
  passByReference(x);
  cout << "x = " << x << endl;
  getchar();
  return 0;
}
```

```
X = 5
X = 7
```

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

void copy_vector(vector<int> v2)
{  v2.at(0) = 2;}

void pass_vector(vector<int> &v3)
{  v3.at(0) = 3;}

int main()
{
vector<int> v;
v.push_back(5);v.push_back(6); v.push_back(7);
vector<int>::iterator it;

copy_vector(v);
for (it = v.begin(); it != v.end(); )
    cout << *it++ << " ";
cout << endl;
pass_vector(v);
for (it = v.begin(); it != v.end(); )
cout << *it++ << " ";
return 0;}
```

When a vector is passed as a parameter to some function, a copy of vector is actually created.

Output: 5 6 7
3 6 7

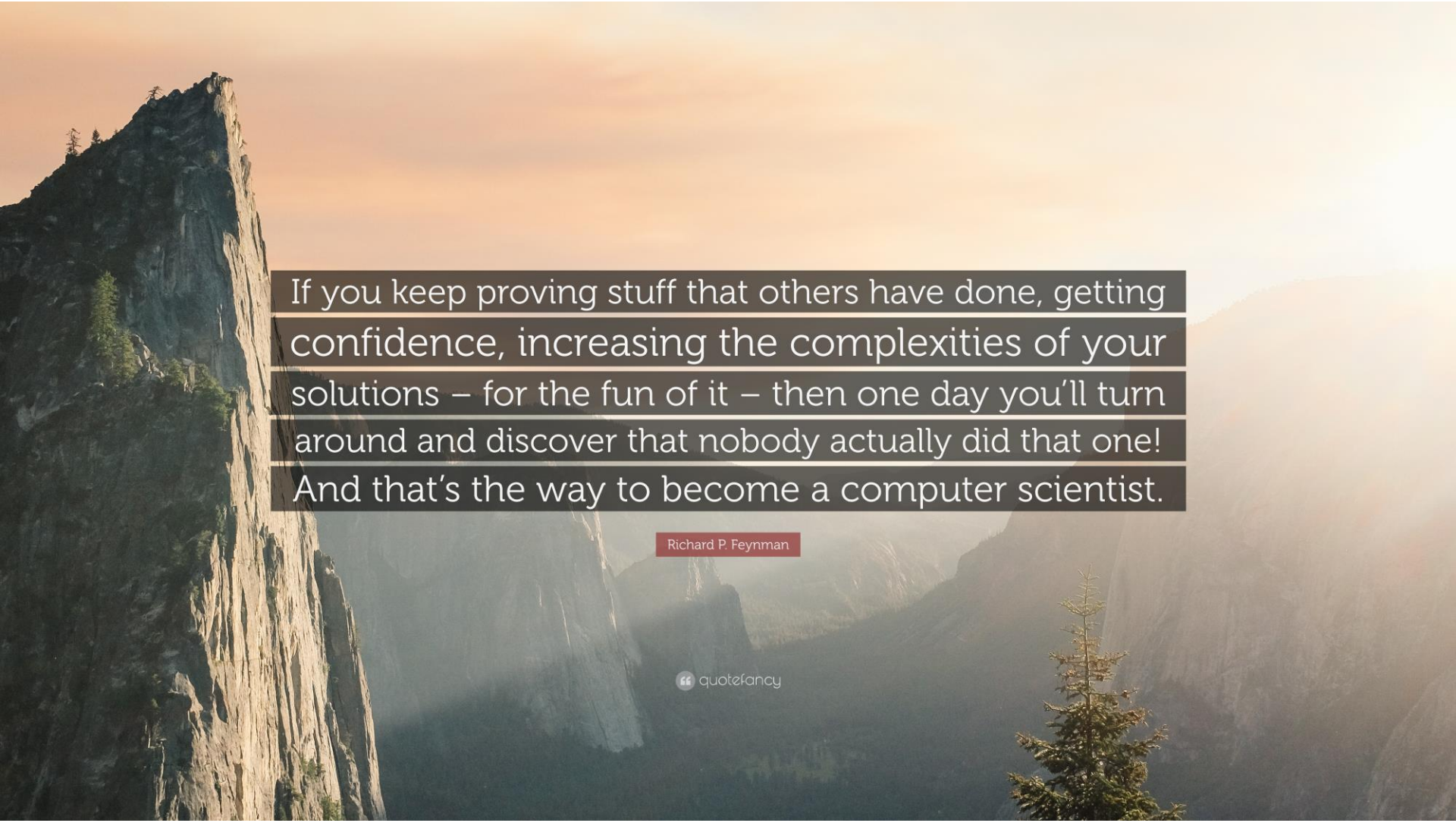
# setw

- **setw (int n);**
- Set field width
- Sets the field width to be used on output operations.

```
#include <iostream>      // std::cout, std::endl
#include <iomanip>       // std::setw

int main() {
std::cout << std::setw(4);
std::cout << 55;
return 0;
}
```

Write a C++ program to enter 10 random numbers between 5 and 9 into a vector. Then call a function removeEven(vector<int>& v) to remove all even numbers. Finally print the vector

```
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;
void removeEven(vector < int > & v2) {
  vector < int > ::iterator it;
  for (it = v2.begin(); it != v2.end();)
      if ( * it % 2 == 0) it = v2.erase(it);
      else it++;}
int main() {
  int random;
  vector < int > v1;
  vector < int > ::iterator it;
  srand(time(NULL));
  for (int i = 0; i < 10; i++) {
       random = 5 + rand() % 5;
       cout << random << " ";
       v1.push_back(random);}
  removeEven(v1);
  cout << "\n After removing even numbers:";
  for (it = v1.begin(); it != v1.end(); it++)
      cout << * it << " ";
  return 0;}
```


If you keep proving stuff that others have done, getting confidence, increasing the complexities of your solutions – for the fun of it – then one day you'll turn around and discover that nobody actually did that one! And that's the way to become a computer scientist.
Richard P. Feynman
quotefancy

# Searching and Sorting Algorithms

- InputIterator **find** (InputIterator first, InputIterator last, const T& val);
  - Returns an iterator to the **first element** in the range [first,last) that compares equal to val. If no such element is found, the function returns **last**.
- InputIterator **find_if** (InputIterator first, InputIterator last, UnaryPredicate pred);
  - Returns an iterator to the first element in the range [first,last) for which pred returns true. If no such element is found, the function returns last.
  - **pred**: Unary function that accepts an element in the range as argument and returns a value convertible to **bool**. The value returned indicates whether the element is considered a match in the context of this function.
- bool **binary_search** (ForwardIterator first, ForwardIterator last, const T& val);
  - Returns true if any element in the range [first,last) is equivalent to val, and false otherwise.
  - The elements in the range shall already be **sorted**.
- void **sort** (RandomAccessIterator first, RandomAccessIterator last);
  - Sorts the elements in the range [first,last) into ascending order.

```cpp
#include <iostream>
#include <set>
using namespace std;
int main() {
set<int> s;
set<int>::iterator it;

for (int i = 1; i <= 9; i++)
     s.insert(i);

s.erase(5);

it = s.begin();
++it;

s.erase(it, s.find(7));

for (it = s.begin(); it != s.end(); ++it)
     cout << *it << " ";

return 0;
}
```

**Output: 1 7 8 9**

```
#include <iostream>
#include <list>
#include <algorithm>
bool IsOdd(int i) {return ((i % 2) == 1);}
using namespace std;
int main()
{
list<int> li;
for (int nCount = 0; nCount < 6; nCount++)
     li.push_back(nCount);

list<int>::const_iterator it;
it = find(li.begin(), li.end(), 3);
li.insert(it, 8);

for (it = li.begin(); it != li.end(); it++)
     cout << *it << " ";

cout<< *(find_if(li.begin(), li.end(), IsOdd));
return 0;
}
```

Output: 0 1 2 8 3 4 5 1

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
using namespace std;
vector<int> vect;
vect.push_back(7);   vect.push_back(-3);
vect.push_back(6);   vect.push_back(2);
vect.push_back(-5);   vect.push_back(0);
sort(vect.begin(), vect.end());

vector<int>::const_iterator it;
for (it = vect.begin(); it != vect.end(); it++)
     cout << *it << " ";
cout << endl;

return 0;
}
```

```
-5 -3 0 2 6 7
```

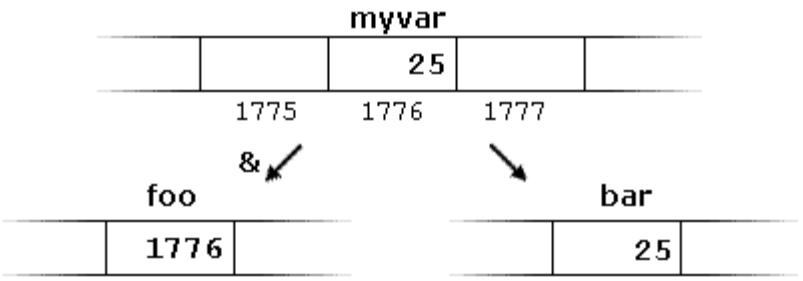
```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;
bool greater10(int value)
{return value > 10;}
int main()
{
const int SIZE = 10;
int a[SIZE] = { 10, 2, 17, 5, 16, 8, 12, 11, 20, 7 };
vector<int> v(a, a + SIZE); // copy of a
vector<int>::iterator location;

location = find(v.begin(), v.end(), 16);
if (location != v.end())
    cout << "Found 16 at location " << (location - v.begin()) << endl;
else
    cout << "16 not found \n";

location = find_if(v.begin(), v.end(), greater10);
if (location != v.end())
    cout << "The first value greater than 10 is " << *location << endl;
else
    cout << "No values greater than 10 were found \n";

if (binary_search(v.begin(), v.end(), 12))
   cout << "12 was found in v \n";
else
   cout << "12 was not found in v \n";
sort(v.begin(), v.end());
if (binary_search(v.begin(), v.end(), 12))
    cout << "12 was found in v \n";
else
    cout << "12 was not found in v \n";
return 0;}
```

```
Found 16 at location 4
The first value greater than 10 is 17
12 was not found in v
12 was found in v
```

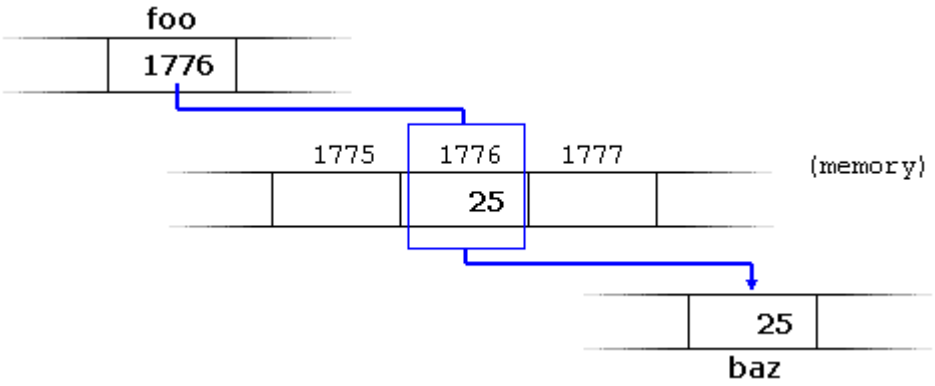
# Pointers

- The declaration of pointers follows this syntax:
  - `type * name;`
  - `int *foo;  //declaring a pointer`
- The variable that stores the address of another variable (like foo in the previous example) is what in C++ is called a **pointer**.
- The address of a variable can be obtained by preceding the name of a variable with an ampersand sign **(&),** known as address-of operator. For example:
  - `foo = &myvar;`
- This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the address-of operator (&), we are no longer assigning the content of the variable itself to foo, but its address.
- More details: http://www.cplusplus.com/doc/tutorial/pointers/

- Assume `myvar` is placed during runtime in the memory address 1776.
  - `myvar = 25;`
  - `foo = &myvar;`
  - `bar = myvar;`




- Pointers can be used to access the variable they point to directly. This is done by preceding the pointer name with the **dereference operator (*)**.
  - `int baz = *foo;`




- Thus, & and * have sort of **opposite** meanings: An address obtained with & can be dereferenced with *.

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

int main()
{
int firstvalue, secondvalue;
int * mypointer;

mypointer = &firstvalue;
*mypointer = 10;
mypointer = &secondvalue;
*mypointer = 20;
cout << "firstvalue is " << firstvalue << '\n';
cout << "secondvalue is " << secondvalue << '\n';
return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

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

int main()
{
int firstvalue = 5, secondvalue = 15;
int * p1, *p2;

p1 = &firstvalue;  // p1 = address of firstvalue
p2 = &secondvalue; // p2 = address of secondvalue
*p1 = 10;          // value pointed to by p1 = 10
*p2 = *p1;         // value pointed to by p2 = value pointed to by p1
p1 = p2;           // p1 = p2 (value of pointer is copied)
*p1 = 20;          // value pointed to by p1 = 20

cout << "firstvalue is " << firstvalue << '\n';
cout << "secondvalue is " << secondvalue << '\n';
return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

# Pointers and arrays

- The concept of arrays is related to that of pointers. In fact**, arrays work very much like pointers** to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:
  - `int myarray [20];`
  - `int * mypointer;`
- The following assignment operation would be valid:
  - `mypointer = myarray;`
- After that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would **not** be valid:
  - `myarray = mypointer;`

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

int main()
{
int numbers[5];
int * p;
p = numbers;  *p = 10;
p++;  *p = 20;
p = &numbers[2];  *p = 30;
p = numbers + 3;  *p = 40;
p = numbers;  *(p + 4) = 50;
for (int n = 0; n<5; n++)
     cout << numbers[n] << ", ";
return 0;
}
```

```
10, 20, 30, 40, 50,
```

# Pointers to functions

- C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the **name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:**

# Pointer to function – example 1

```
#include <iostream>
using namespace std;
void one(int a, int b) { cout << a + b << "\n"; }
void two(int a, int b) { cout << a*b << "\n"; }

int main()
{
void(*fptr)(int, int); // a function pointer to voids with two
int params

fptr = one; //fptr -> one
fptr(12, 3); //=> one(12, 3)

fptr = two; //fptr -> two
fptr(5, 4); //=> two(5, 3)

return 0;}
```

```
15
20
```

# Pointer to function – example 2

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

int add(int first, int second)
{return first + second;}

int subtract(int first, int second)
{return first - second;}

int operation(int first, int second,
int(*functocall)(int, int))
{return functocall(first, second);}

int main()
{
int  a, b;
a = operation(7, 5, add);
b = operation(20, a, subtract);
cout << "a = " << a << " and b = " << b << endl;
return 0; }
```

Output: a = 12 and b = 8

# Functions to determine the **size** of a vector container

| Expression | Effect |
|---|---|
| `vecCont.capacity()` | Returns the maximum number of elements that can be inserted into the container `vecCont` without reallocation. |
| `vecCont.empty()` | Returns **`true`** if the container `vecCont` is empty and **`false`** otherwise. |
| `vecCont.size()` | Returns the number of elements currently in the container `vecCont`. |
| `vecCont.max_size()` | Returns the maximum number of elements that can be inserted into the container `vecCont`. |

```cpp
// comparing size, capacity and max_size
#include <iostream>
#include <vector>

int main()
{
std::vector<int> myvector;

// set some content in the vector:
for (int i = 0; i<100; i++) myvector.push_back(i);

std::cout << "size: " << myvector.size() << '\n';
std::cout << "capacity: " << myvector.capacity() << '\n';
std::cout << "max_size: " << myvector.max_size() << '\n';
return 0;
}
```

```
size: 100
capacity: 141
max_size: 1073741823
```

# Member Functions Common to **All** Containers

- Examples
  - Default constructor
  - Several constructors with parameters
  - Destructor
  - Function inserting an element into a container
- Class encapsulates data, operations on that data
  - Into a single unit
- Every container is a **class**
  - Several operations directly defined for a container
  - Provided as part of class definition

# Member functions common to **all** containers

| Member function | Effect |
|---|---|
| Default constructor | Initializes the object to an empty state. |
| Constructor with parameters | In addition to the default constructor, every container has constructors with parameters. We describe these constructors when we discuss a specific container. |
| Copy constructor | Executes when an object is passed as a parameter by value, and when an object is declared and initialized using another object of the same type. |
| Destructor | Executes when the object goes out of scope. |
| `ct.empty()` | Returns `true` if container `ct` is empty and `false` otherwise. |
| `ct.size()` | Returns the number of elements currently in container `ct`. |
| `ct.max_size()` | Returns the maximum number of elements that can be inserted into container `ct`. |
| `ct1.swap(ct2)` | Swaps the elements of containers `ct1` and `ct2`. |
| `ct.begin()` | Returns an iterator to the first element into container `ct`. |
| `ct.end()` | Returns an iterator to ~~the last element in~~into container `ct`. |
| `ct.rbegin()` | Reverse begin. Returns a pointer to the last element into container `ct`. This function is used to process the elements of `ct` in reverse. |
| `ct.rend()` | Reverse end. Returns a pointer to the first element into container `ct`. |
| `ct.insert(position, elem)` | Inserts `elem` into container `ct` at the position specified by the argument `position`. Note that here `position` is an iterator. |
| `ct.erase(begin, end)` | Deletes all elements between `begin...end-1` from container `ct`. |

the element past the end

# Member functions common to **all** containers

| Member function | Effect |
|---|---|
| `ct.clear()` | Deletes all elements from the container. After a call to this function, container `ct` is empty. |
| **Operator functions** | |
| `ct1 = ct2` | Copies the elements of `ct2` into `ct1`. After this operation, the elements in both containers are the same. |
| `ct1 == ct2` | Returns `true` if containers `ct1` and `ct2` are equal and `false` otherwise. |
| `ct1 != ct2` | Returns `true` if containers `ct1` and `ct2` are not equal and `false` otherwise. |
| `ct1 < ct2` | Returns `true` if container `ct1` is less than container `ct2` and `false` otherwise. |
| `ct1 <= ct2` | Returns `true` if container `ct1` is less than or equal to container `ct2` and `false` otherwise. |
| `ct1 > ct2` | Returns `true` if container `ct1` is greater than container `ct2` and `false` otherwise. |
| `ct1 >= ct2` | Returns `true` if container `ct1` is greater than or equal to container `ct2` and `false` otherwise. |

Compares the content

http://en.cppreference.com/w/cpp/container/vector/operator_cmp

# Member Functions Common to Sequence Containers

| Expression | Effect |
|---|---|
| `seqCont.insert(position, elem)` | A copy of `elem` is inserted at the position specified by `position`. The position of the new element is returned. |
| `seqCont.insert(position, n, elem)` | `n` copies of `elem` are inserted at the position specified by `position`. |
| `seqCont.insert(position, beg, end)` | A copy of the elements, starting at `beg` until `end-1`, are inserted into `seqCont` at the position specified by `position`. |
| `seqCont.push_back(elem)` | A copy of `elem` is inserted into `seqCont` at the end. |
| `seqCont.pop_back()` | Deletes the last element. |
| `seqCont.erase(position)` | Deletes the element at the position specified by `position`. |
| `seqCont.erase(beg, end)` | Deletes all elements starting at `beg` until `end-1`. |
| `seqCont.clear()` | Deletes all elements from the container. |
| `seqCont.resize(num)` | Changes the number of elements to `num`. If `size()` grows, the new elements are created by their default constructor. |
| `seqCont.resize(num, elem)` | Changes the number of elements to `num`. If `size()` grows, the new elements are copies of `elem`. |

# Sequence Container: `deque`

- **Deque: double-ended queue**
- Implemented as **dynamic arrays**
- Can expand in **either direction**
- Therefore, they provide a functionality similar to **vectors**, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end.
- Both vectors and deques provide a very similar interface and can be used for similar purposes, but **internally** both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators).
- For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, **deques perform worse** and have less consistent iterators and references than lists.
- Header file `deque` contains
  - Definition of the class `deque`
  - Functions to implement various operations on a `deque` object

# Various ways to declare a `deque` object

| Statement | Effect |
|---|---|
| `deque<elementType> deq;` | Creates an empty `deque` container without any elements. (The default constructor is invoked.) |
| `deque<elementType> deq(otherDeq);` | Creates a deque container, `deq`, and initializes `deq` to the elements of `otherDeq`; `deq` and `otherDeq` are of the same type. |
| `deque<elementType> deq(size);` | Creates a deque container, `deq`, of size `size`. `deq` is initialized using the default constructor. |
| `deque<elementType> deq(n, elem);` | Creates a deque container, `deq`, of size `n`. `deq` is initialized using `n` copies of the element `elem`. |
| `deque<elementType> deq(begin, end);` | Creates a deque container, `deq`. `deq` is initialized to the elements in the range `[begin, end)`—that is, all elements in the range `begin...end-1`. |

# Various operations that can be performed on a `deque` object

| Expression | Effect |
|---|---|
| `deq.assign(n,elem)` | Assigns `n` copies of `elem`. |
| `deq.assign(beg,end)` | Assigns all the elements in the range `beg...end-1`. |
| `deq.push_front(elem)` | Inserts `elem` at the beginning of `deq`. |
| `deq.pop_front()` | Removes the first element from `deq`. |
| `deq.at(index)` | Returns the element at the position specified by `index`. |
| `deq[index]` | Returns the element at the position specified by `index`. |
| `deq.front()` | Returns the first element. (Does not check whether the container is empty.) |
| `deq.back()` | Returns the last element. (Does not check whether the container is empty.) |

```cpp
// vector assign
#include <iostream>
#include <vector>
int main()
{
std::vector<int> first;
std::vector<int> second;
std::vector<int> third;
first.assign(7, 100);             // 7 ints with a value of 100

std::vector<int>::iterator it;
it = first.begin() + 1;

second.assign(it, first.end() - 1); // the 5 central values of first

int myints[] = { 1776,7,4 };
third.assign(myints, myints + 3);   // assigning from array.

std::cout << "Size of first: " << int(first.size()) << '\n';
std::cout << "Size of second: " << int(second.size()) << '\n';
std::cout << "Size of third: " << int(third.size()) << '\n';
return 0;
}
```

```
Size of first: 7
Size of second: 5
Size of third: 3
```

```
// deque::push_front
#include <iostream>
#include <deque>

int main()
{
std::deque<int> mydeque(2, 100);
mydeque.push_front(200);
mydeque.push_front(300);

std::cout << "mydeque contains:";
for (std::deque<int>::iterator it = mydeque.begin(); it
!= mydeque.end(); ++it)
std::cout << ' ' << *it;
std::cout << '\n';

return 0;
}
```

```
mydeque contains: 300 200 100 100
```

# Sequence Container: `list`

- Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and **iteration in both directions**.
- A list is a special type of sequence container called a **doubly linked list** where each element in the container contains pointers that point at the next and previous elements in the list.
- Lists only provide access to the start and end of the list -- **there is no random access provided.**

# Iterating through a vector

```
#include <iostream>
#include <vector>
int main()
{
using namespace std;
vector<int> vect;
for (int nCount = 0; nCount < 6; nCount++)
     vect.push_back(nCount);

vector<int>::const_iterator it;
it = vect.begin();
while (it != vect.end())
       cout << *it++ << " ";
return 0;
}
```

Output: 0 1 2 3 4 5

# Iterating through a deque

```
#include <iostream>
#include <deque>
int main()
{
using namespace std;

deque<int> deq;
for (int nCount = 0; nCount < 3; nCount++)
    {
     deq.push_back(nCount);
     deq.push_front(10 - nCount);
    }

for (int nIndex = 0; nIndex < deq.size(); nIndex++)
     cout << deq[nIndex] << " ";
return 0;
}
```

Output: 8 9 10 0 1 2

# Iterating through a list

```
#include <iostream>
#include <list>
int main()
{
using namespace std;
list<int> li;
for (int nCount = 0; nCount < 6; nCount++)
     li.push_back(nCount);

list<int>::const_iterator it;
it = li.begin();
while (it != li.end())
       cout << *it++ << " ";

return 0;}
```

Output: 0 1 2 3 4 5

Note the code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

```
// inserting into a list
#include <iostream>
#include <list>
#include <vector>
int main()
{
std::list<int> mylist;
std::list<int>::iterator it;

for (int i = 1; i <= 5; ++i) mylist.push_back(i);

it = mylist.begin();
++it;
mylist.insert(it, 10);
mylist.insert(it, 2, 20);
--it;
std::vector<int> myvector(2, 30);
mylist.insert(it, myvector.begin(), myvector.end());
std::cout << "mylist contains:";
for (it = mylist.begin(); it != mylist.end(); ++it)
std::cout << ' ' << *it;
std::cout << '\n';
return 0;}
```

```
mylist contains: 1 10 20 30 30 20 2 3 4 5
```

# list operations

- **void remove (const value_type& val);**
  - remove all elements with specific value
- **Sort**
  - Sorts the elements in the list, altering their position within the container.
  - **void sort();**
    - Sorts the elements in the list, altering their position within the container.
  - **void sort (Compare comp);**

```
#include <iostream>
#include <list>
#include <string>
#include <cctype>
bool compare_nocase(const std::string& first, const std::string&
second)
{
unsigned int i = 0;
while ((i<first.length()) && (i<second.length()))
{
 if (tolower(first[i])<tolower(second[i])) return true;
 else if (tolower(first[i])>tolower(second[i])) return false;
 ++i;
}
return (first.length() < second.length());
}
```

```
int main()
{
std::list<std::string> mylist;
std::list<std::string>::iterator it;
mylist.push_back("one");
mylist.push_back("two");
mylist.push_back("Three");

mylist.sort();

std::cout << "mylist contains:";
for (it = mylist.begin(); it != mylist.end(); ++it) std::cout <<
' ' << *it;
std::cout << '\n';

mylist.sort(compare_nocase);

std::cout << "mylist contains:";
for (it = mylist.begin(); it != mylist.end(); ++it) std::cout <<
' ' << *it;
std::cout << '\n'; return 0; }
```

```
mylist contains: Three one two
mylist contains: one Three two
```

# list operations - Merge

- **void merge (list& x);**
  - Merges x into the list by transferring all of its elements at their respective ordered positions into the container.
  - This effectively **removes all** the elements in x (which becomes **empty**), and inserts them into their ordered position within container (which expands in size by the number of elements transferred).
  - This function requires that the list containers have their elements already **ordered** by value (or by comp) before the call.
- **void merge (list& x, Compare comp);**
  - Have the same behavior, but take a specific predicate (comp) to perform the comparison operation between elements.

```
#include <iostream>
#include <list>
bool mycomparison(double first, double second)
{return ((first)<(second));}
int main()
{
std::list<double> first, second;
first.push_back(3.1); first.push_back(2.2); first.push_back(2.9);
second.push_back(3.7); second.push_back(7.1); second.push_back(1.4);
first.sort(); second.sort();
first.merge(second);

std::cout << "first contains: ";
for (std::list<double>::iterator it = first.begin(); it != first.end(); ++it)
std::cout << *it << ",  ";
std::cout << '\n';

second.push_back(2.1);
first.merge(second, mycomparison);
std::cout << "Now first contains: ";
for (std::list<double>::iterator it = first.begin(); it != first.end(); ++it)
std::cout << *it << ",  ";
return 0;}
```

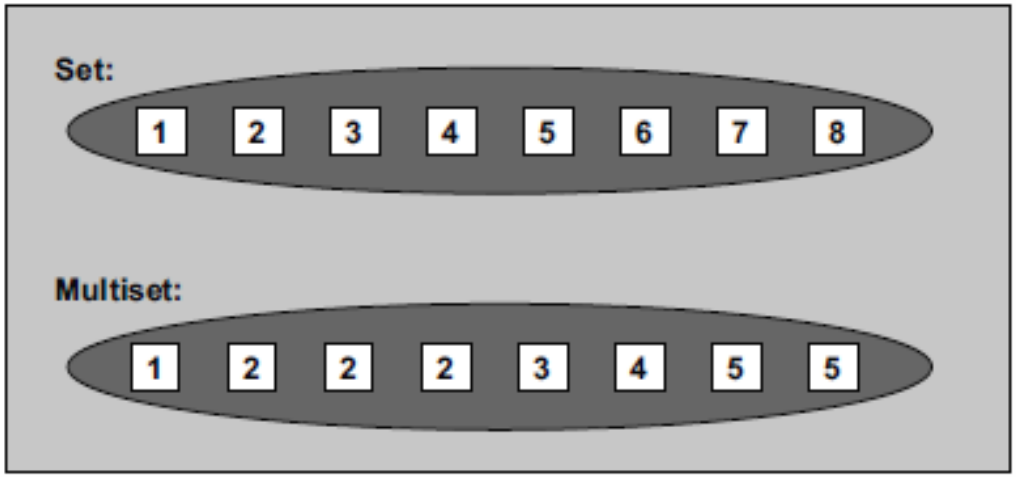
```
first contains: 1.4,  2.2,  2.9,  3.1,  3.7,  7.1,
first contains: 1.4,  2.1,  2.2,  2.9,  3.1,  3.7,  7.1,
```

# Associative Containers

- Associative contains are containers that **automatically sort** their inputs when those inputs are inserted into the container. By default, associative containers compare elements using operator< (less than).
- Elements in associative containers are referenced by their **key** and not by their absolute position in the container.
- A **set** is a container that stores **unique** elements.
- A **multiset** is a set where **duplicate** elements are allowed.
- A **map** (also called an associative array) is a set where each element is a pair, called a **key/value** pair. The key is used for sorting and indexing the data, and must be unique. The value is the actual data.
- A **multimap** (also called a dictionary) is a map that allows **duplicate** keys. Real-life dictionaries are multimaps: the key is the word, and the value is the meaning of the word.

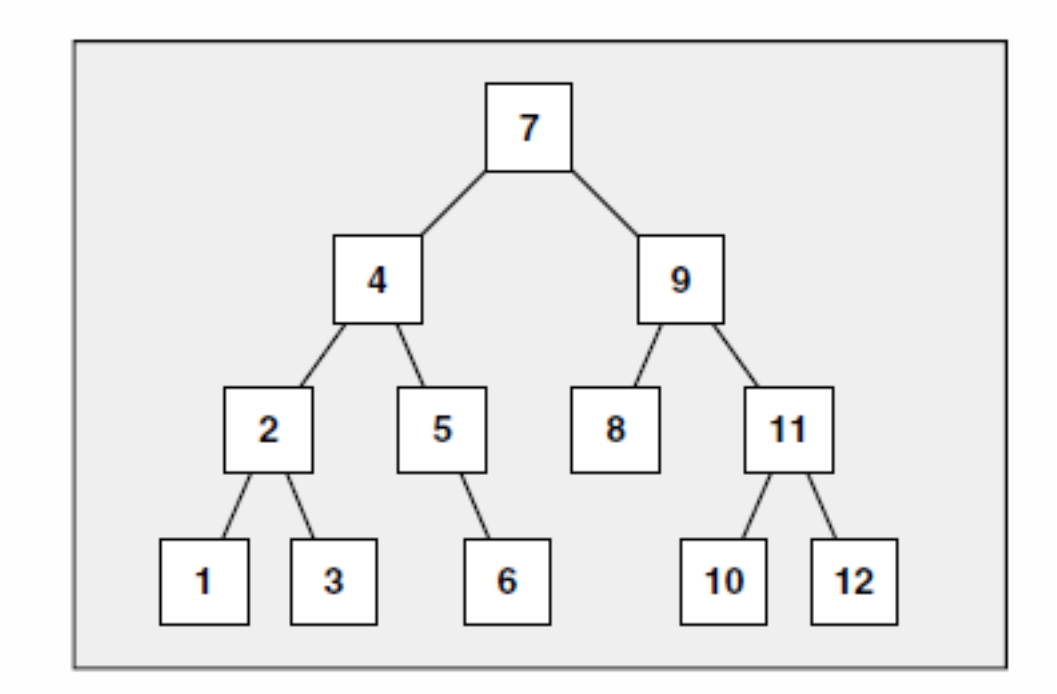
# Sets

- Sets are containers that store **unique** elements following a specific **order**.
- Multisets allow duplicates.
- To use a set or a multiset, you must include the header file **<set>.**



- Sets are typically implemented as **binary search trees.**

Internal Structure of Sets and Multisets



- The value of the elements in a set cannot be modified once in the container (the elements are always **const**), because doing so might compromise the correct **order,** but they can be inserted or removed from the container.
- As with all associative container classes, the iterators are **bidirectional iterators.**

- If you want to use a **sort** criterion other than the default, you must specify this option when the container is declared.
- Include header file <**functional**>
- `set<int> intSet; //ascending order`
- `set<int, greater<int> > otherIntSet; //descending order`
- `multiset<string> stringMultiSet;`
- `multiset<string, less<string> > otherStringMultiSet;`

# Operations to insert elements in a set

**(1) mySet.insert(val)**

- Inserts a copy of val into mySet.

**(2) mySet.insert(iteratorPos, val)**

- Inserts a copy of val into mySet.
- The position where val is inserted is returned.
- The first parameter, **hints** at where to begin the search for insert.

**(3) mySet.insert(iteratorBegin, iteratorEnd);**

- Inserts a copy of all the elements into mySet starting at the position iteratorBegin until iteratorEnd-1.

```
#include <iostream>
#include <set>
#include <functional>
using namespace std;
int main() {
set<int, less<int>> s1;    //ascending order
set<int, greater<int>> s2; //descending order
set<int>::iterator it;
for (int i = 7; i <= 9; i++)
{s1.insert(i);
 s2.insert(i * 10);}

for (it = s1.begin(); it != s1.end(); ++it)
cout << *it << " ";
cout << endl;
for (it = s2.begin(); it != s2.end(); ++it)
cout << *it << " ";
return 0;}
```

**Output: 7 8 9**
**90 80 70**

```
#include <iostream>
#include <set>
#include <vector>
using namespace std;
int main() {
vector<int> v;
set<int> s;
set<int>::iterator it;

v.push_back(2);
v.push_back(10);

for (int i = 7; i <= 9; i++)
s.insert(i);

s.insert(v.begin(), v.end());

for (set<int>::iterator it = s.begin(); it != s.end(); ++it)
cout << *it << " ";
return 0;}
```

**Output: 2 7 8 9 10**

# Operations to remove elements from a set

**(1) mySet.erase(val);**

- Deletes all the elements with the value val.
- The number of deleted elements is returned.

**(2) mySet.erase(iteratorPos);**

- Deletes the element at the position specified by the iterator position.
- **Return an iterator to the element that follows the last element removed (or set::end, if the last element was removed).**

**(3) mySet.erase(iteratorBegin, iteratorEnd);**

- Deletes all the elements starting at the position iteratorBegin **until iteratorEnd-1.**

**(4) mySet.clear();**

- Deletes all the elements from mySet.
- After this operation, mySet is empty.

- **mySet.size()**
  - Returns the current number of elements
- **mySet.count(val)**
  - Returns the number of elements with value val

# Iterating through a set

```
#include <iostream>
#include <set>
int main()
{
using namespace std;
set<int> myset;
myset.insert(7);  myset.insert(2);
myset.insert(-6); myset.insert(8);
myset.insert(1);  myset.insert(-4);

set<int>::const_iterator it;
it = myset.begin();
while (it != myset.end())
      cout << *it++ << " ";

cout << endl;
return 0;
}
```

Output: -6 -4 1 2 7 8

# Iterating through a map

```
#include <iostream>
#include <map>
#include <string>
int main()
{
using namespace std;
map<int, string> mymap;
mymap.insert(make_pair(4, "apple"));
mymap.insert(make_pair(1, "orange"));
mymap.insert(make_pair(3, "grapes"));
mymap.insert(make_pair(2, "peach"));

map<int, string>::const_iterator it;
it = mymap.begin();
while (it != mymap.end())
      {cout << it->first << "=" << it->second << " ";
      it++;}
return 0;}
```

Output: 1=orange 2=peach 3=grapes 4=apple

# Sieve of Eratosthenes (Siv of air-uh Taws-thuh neeze)

- A simple algorithm to find **prime numbers** from 2 to N
- Examples
  - http://www.algolist.net/Algorithms/Number_theoretic/Sieve_of_Eratosthenes
  - http://www.visnos.com/demos/sieve-of-eratosthenes
  - https://www.youtube.com/watch?v=V08g_lkKj6Q
  - https://www.youtube.com/watch?v=9m2cdWorIq8

# Iterators

- An iterator is any object that points to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators).
- It represents a **position** in the container
- A **pointer** is a form of iterator.
- Iterators must be implemented on a per-class basis, because the iterator does need to know how a class is implemented. Thus iterators are always tied to **specific container** classes.
  - Each STL container defines what iterators it can return.
- Instead of operating on specific data types, **algorithms** are defined to operate on a range specified by a type of iterator.
  - Each algorithm specifies what class of iterators it requires.
- Iterators can be **generated** by STL container member functions, such as begin() and end().

- All STL containers (but not the adapters) provide at least two types of iterators:

**(1) `iterator`**

- Every container (sequence or associative) contains an `iterator`
- **Read/Write** iterator

**(2) `const_iterator`**

- Prevents iterator from modifying elements of container declared as constant
- Every container contains `const_iterator`
- **Read-only** iterator

# Iterator categories

1. Input iterators
2. Output iterators
3. Forward iterators
4. Bidirectional iterators
5. Random access iterators

# (1) Input Iterators

- **Read** data from an input stream
- Step **forward** element-by-element
- Return values element-by-element
- Input iterators can read elements **only once**.
- Example
  - `InputIterator` **`find`** `(`**`InputIterator`** `first,` **`InputIterator`** `last, const T& val);`

# Operations on an input iterator

| Expression | Effect |
|---|---|
| `*inputIterator` | Gives access to the element to which `inputIterator` points. |
| `inputIterator->member` | Gives access to the member of the element. |
| `++inputIterator` | Moves forward, returns the new position (preincrement). |
| `inputIterator++` | Moves forward, returns the old position (postincrement). |
| `inputIt1 == inputIt2` | Returns `true` if the two iterators are the same and `false` otherwise. |
| `inputIt1 != inputIt2` | Returns `true` if the two iterators are not the same and `false` otherwise. |

# (2) Output Iterators

- **Write** data to an output stream
- Step forward element-by-element
- As with input iterators, you can't use an output iterator to iterate twice over the same range.
- Example
  - `OutputIterator` **`copy`** `(InputIterator first, InputIterator last,` **`OutputIterator`** `result);`

# Operations on an output iterator

| Expression | Effect |
| --- | --- |
| `*outputIterator = value;` | Writes the `value` at the position specified by the `outputIterator`. |
| `++outputIterator` | Moves forward, returns the new position (preincrement). |
| `outputIterator++` | Moves forward, returns the old position (postincrement). |

# (3) Forward Iterators

- Combines the functionality of the **input** and **output** iterators.
- Unlike input and output iterators, forward iterators can refer to the same element in the same collection and process the same element more than once (can be used in **multipass** algorithms).
- Example
  - `bool` **`binary_search`** `(`**`ForwardIterator`** `first, ForwardIterator last, const T& val);`

# Operations on a forward iterator

| Expression | Effect |
|---|---|
| `*forwardIterator` | Gives access to the element to which `forwardIterator` points. |
| `forwardIterator->member` | Gives access to the member of the element. |
| `++forwardIterator` | Moves forward, returns the new position (preincrement). |
| `forwardIterator++` | Moves forward, returns the old position (postincrement). |
| `forwardIt1 == forwardIt2` | Returns `true` if the two iterators are the same and `false` otherwise. |
| `forwardIt1 != forwardIt2` | Returns `true` if the two iterators are not the same and `false` otherwise. |
| `forwardIt1 = forwardIt2` | Assignment. |

# (4) Bidirectional Iterators

- Allow algorithms to pass through the elements **forward** and **backward**.
- Operations defined for **forward iterators** applicable to bidirectional Iterators
- This type of iterator can used with the sequence and associative containers.
- Additional operations on a bidirectional iterator

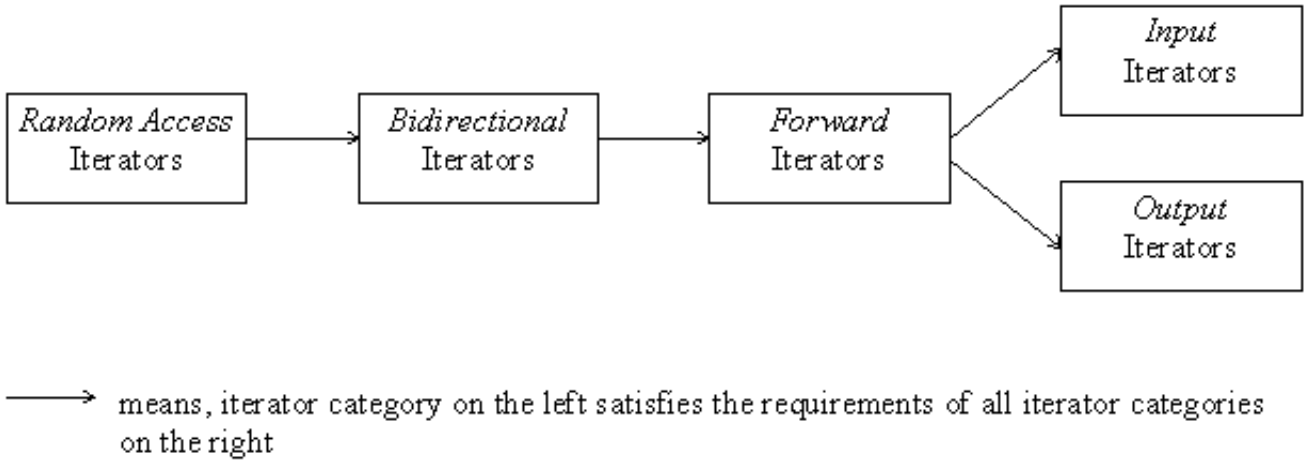
| Expression | Effect |
|---|---|
| `--biDirectionalIterator` | Moves backward, returns the new position (predecrement). |
| `biDirectionalIterator--` | Moves backward, returns the old position (postdecrement). |

# (5) Random Access Iterators

- Bidirectional iterators that can **randomly** process container elements
- Can be used with containers of type:
  - `vector` , `deque` , `string`, and `arrays`
- Operations defined for bidirectional iterators applicable to random access iterators
  - `void` **`sort`** `(`**`RandomAccessIterator`** `first, RandomAccessIterator last);`

# Additional operations on a random access iterator

| Expression | Effect |
|---|---|
| `rAccessIterator[n]` | Accesses the nth element. |
| `rAccessIterator += n` | Moves `rAccessIterator` forward `n` elements if `n >= 0` and backward if `n < 0`. |
| `rAccessIterator -= n` | Moves `rAccessIterator` backward `n` elements if `n >= 0` and forward if `n < 0`. |
| `rAccessIterator + n` | Returns the iterator of the next nth element. |
| `n + rAccessIterator` | Returns the iterator of the next nth element. |
| `rAccessIterator - n` | Returns the iterator of the previous nth element. |
| `rAccessIt1 - rAccessIt2` | Returns the distance between the iterators `rAccessIt1` and `rAccessIt2`. |
| `rAccessIt1 < rAccessIt2` | Returns `true` if `rAccessIt1` is before `rAccessIt2` and `false` otherwise. |
| `rAccessIt1 <= rAccessIt2` | Returns `true` if `rAccessIt1` is before or equal to `rAccessIt2` and `false` otherwise. |
| `rAccessIt1 > rAccessIt2` | Returns `true` if `rAccessIt1` is after `rAccessIt2` and `false` otherwise. |
| `rAccessIt1 >= rAccessIt2` | Returns `true` if `rAccessIt1` is after or equal to `rAccessIt2` and `false` otherwise. |




- **Input and output iterators** are the most limited types of iterators: they can perform sequential single-pass input or output operations.
- All forward, bidirectional and random-access iterators are also valid **input iterators**.
- This arrangement means that a template function which expects for example a **bidirectional** iterator can be provided with a **random access** iterator, but never with a forward iterator.

| category | | | | properties | valid expressions |
|---|---|---|---|---|---|
| all categories | | | | *copy-constructible*, *copy-assignable* and *destructible* | `X b(a);`<br>`b = a;` |
| | | | | Can be incremented | `++a`<br>`a++` |
| Random Access | Bidirectional | Forward | Input | Supports equality/inequality comparisons | `a == b`<br>`a != b` |
| | | | | Can be dereferenced as an *rvalue* | `*a`<br>`a->m` |
| | | | Output | Can be dereferenced as an *lvalue* (only for *mutable iterator types*) | `*a = t`<br>`*a++ = t` |
| | | | | *default-constructible* | `X a;`<br>`X()` |
| | | | | Multi-pass: neither dereferencing nor incrementing affects dereferenceability | `{ b=a; *a++; *b; }` |
| | | | | Can be decremented | `--a`<br>`a--`<br>`*a--` |
| | | | | Supports arithmetic operators + and - | `a + n`<br>`n + a`<br>`a - n`<br>`a - b` |
| | | | | Supports inequality comparisons (<, >, <= and >=) between iterators | `a < b`<br>`a > b`<br>`a <= b`<br>`a >= b` |
| | | | | Supports compound assignment operations += and -= | `a += n`<br>`a -= n` |
| | | | | Supports offset dereference operator ([]) | `a[n]` |

# != and <

- To write generic code for arbitrary containers, you should use **!=** operator rather than **<** operator. The following loop works with **any container**:
  - for(pos = contner.begin(); pos != contner.end(); ++pos) {...}
- Operator **<** is only provided for random access iterators, so it doesn't work with lists, sets, and maps. The following does not work with all containers:
  - for(pos = contner.begin(); pos < contner.end(); ++pos) {...}

# The `copy` Algorithm

- **copy** (**InputIterator** first, **InputIterator** last, **OutputIterator** result);
- Copies the elements in the range [first,last) into the range beginning at result.
- **copy assumes** that the destination already has room for the elements being copied. It would be an error to copy into an empty list or vector. However, this limitation is easily overcome with insert operators.
- Contained in header file `algorithm`

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
int arr[] = { 1,2,3 };
vector<int> v(arr, arr + sizeof(arr) / sizeof(arr[0])), v1(3);
vector<int>::iterator it;

copy(v.begin(), v.end(), v1.begin()+1);

for (it = v1.begin(); it != v1.end(); it++)
cout << *it << " ";

return 0;
}
```

Output: 0 1 2