Standard Template Library (STL)

The C++ STL

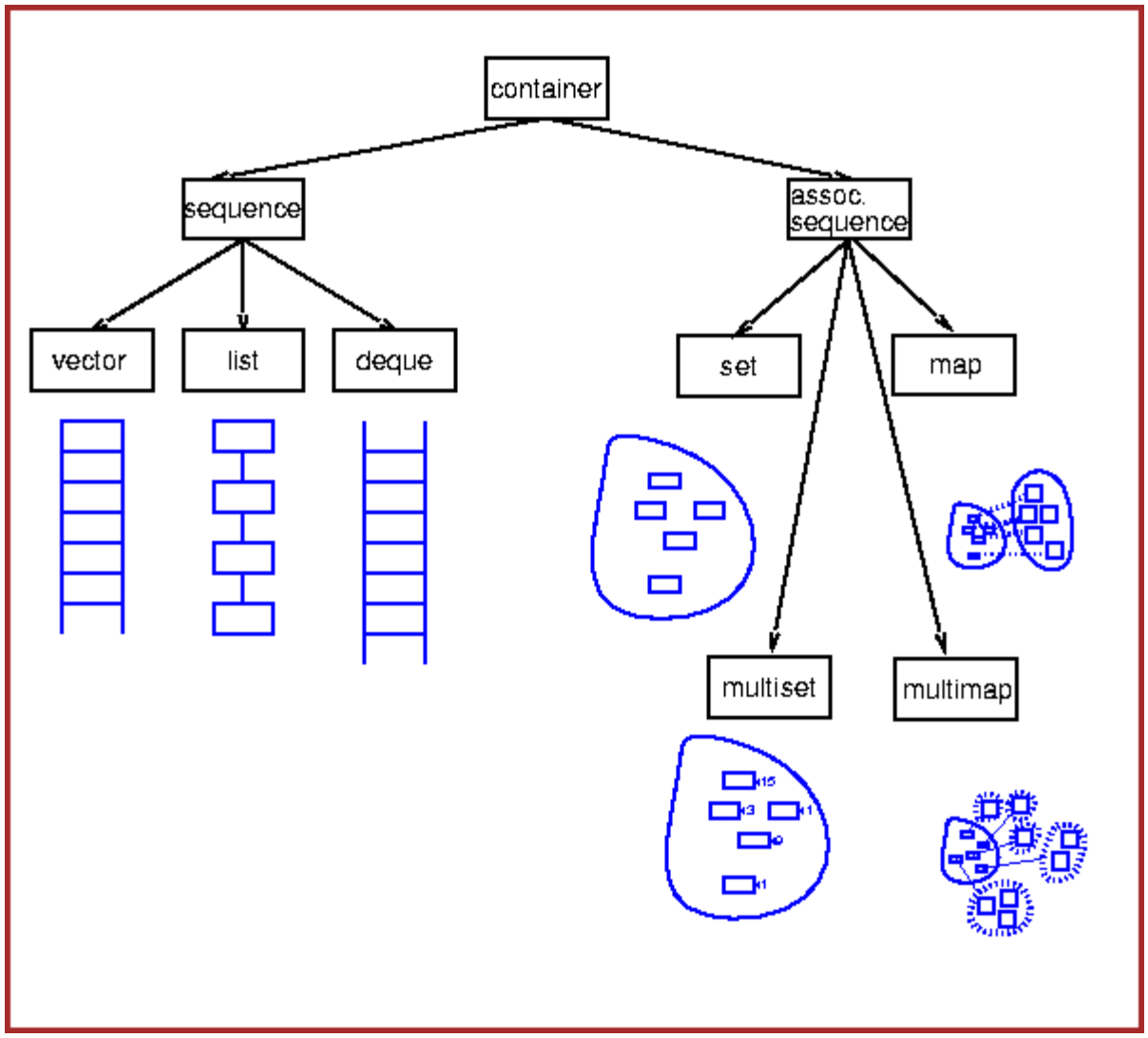
- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.

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

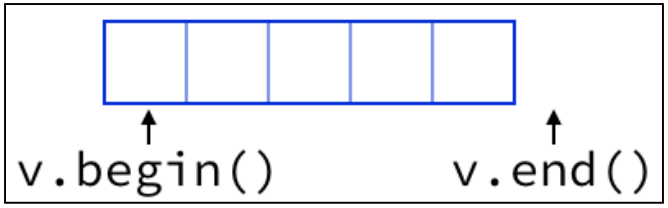
```
vector<int>::iterator intVecIter;
```

- 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

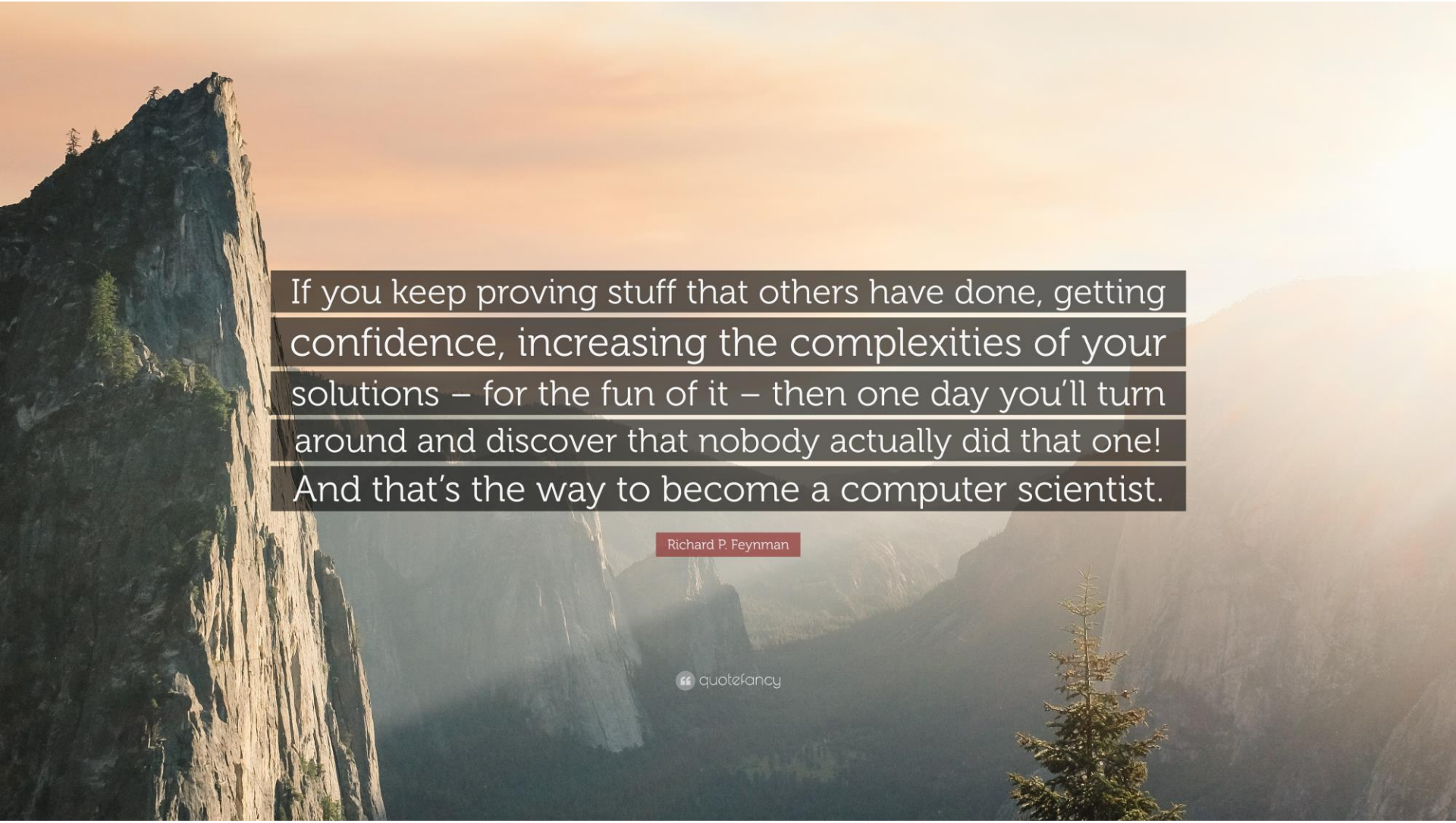
setw

- **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
```

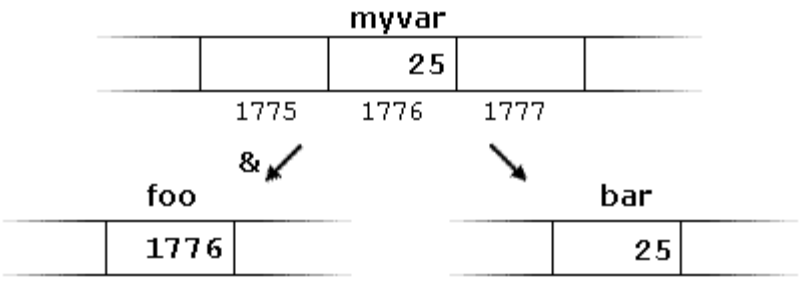
```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;
bool greater10(int value)
{return value > 10;}
int main()
{
const int SIZE = 10;
int a[SIZE] = { 10, 2, 17, 5, 16, 8, 12, 11, 20, 7 };
vector<int> v(a, a + SIZE); // copy of a
vector<int>::iterator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

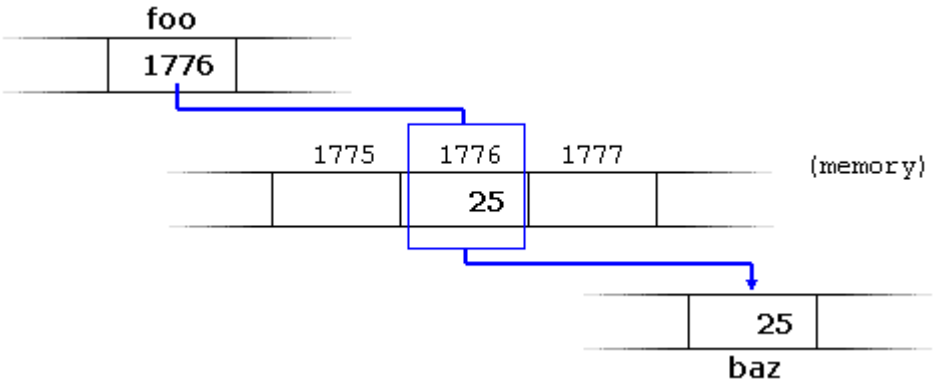
Pointers

- 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