

Standard Template Library (STL)

The C++ STL

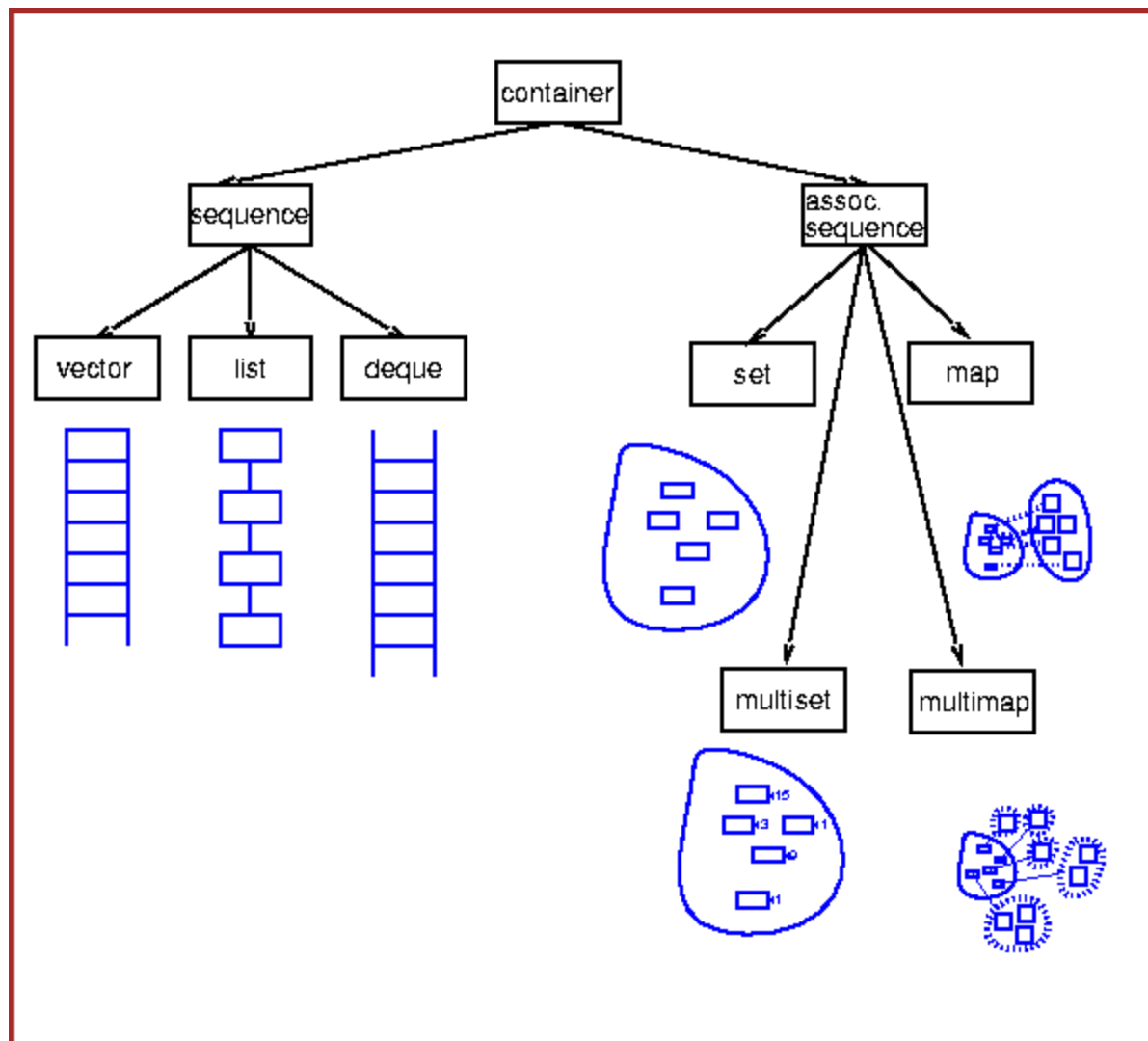
- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.

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**.



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

– Examples:

- `vector<int> intlist;`
- `vector<string> stringList;`

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

Expression	Effect
<code>vecList.at(index)</code>	Returns the element at the position specified by <code>index</code> .
<code>vecList[index]</code>	Returns the element at the position specified by <code>index</code> .
<code>vecList.front()</code>	Returns the first element. (Does not check whether the container is empty.)
<code>vecList.back()</code>	Returns the last element. (Does not check whether the container is empty.)

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

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

```
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;}

```

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

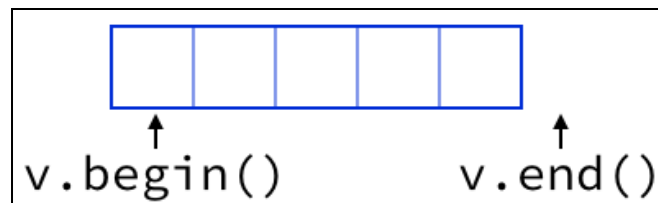
- class vector **contains** typedef iterator
 - Declared as a public member
 - Vector container iterator
 - Example

```
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`.



```
#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

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

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

```
// 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

```
#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

```
#include <iostream>
```

```
#include <vector>
```

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

```
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;}
```

The `sort` Algorithm

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

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

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

```
// 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

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

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

```
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;}

```

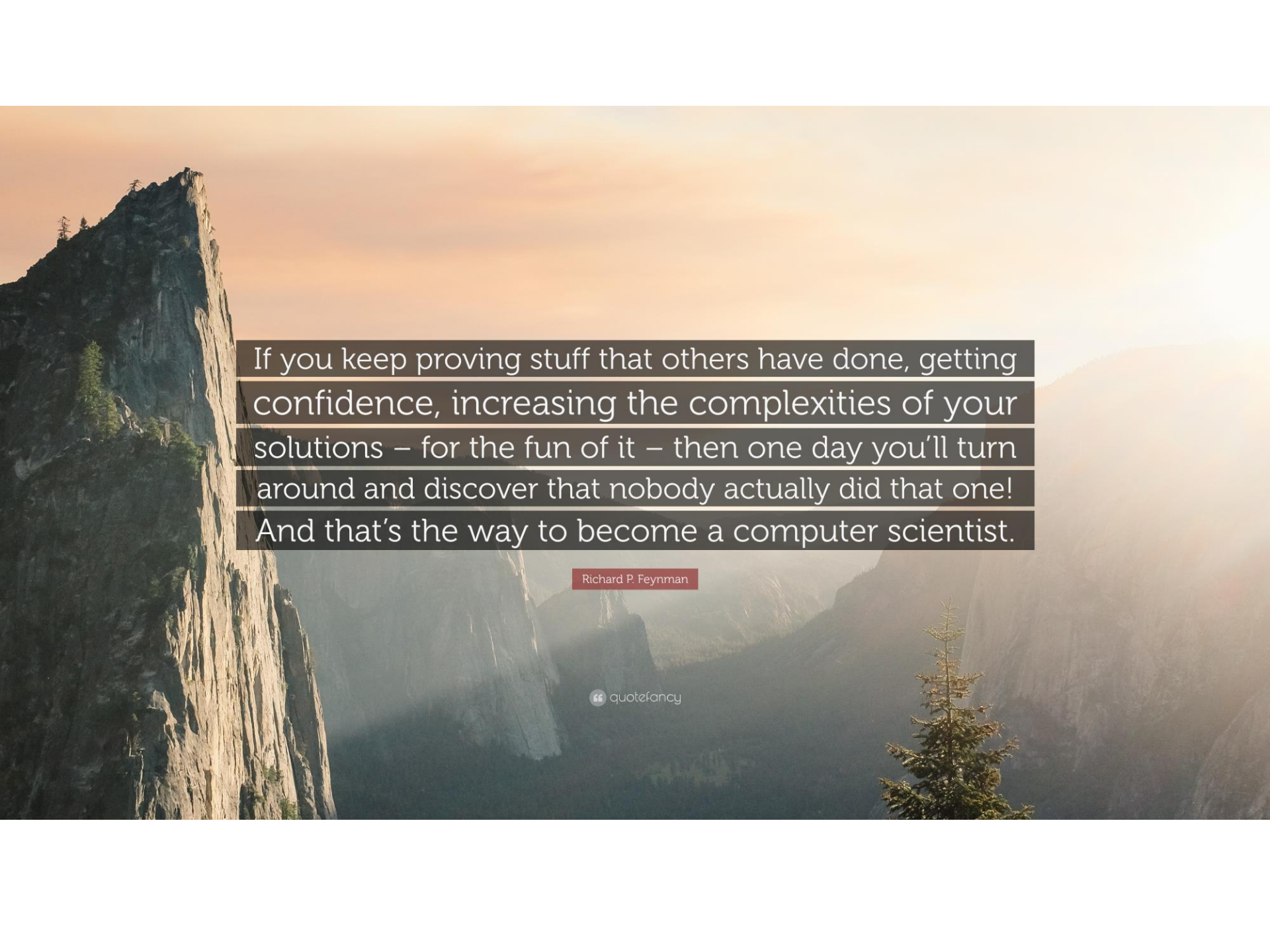
Output: 5 6 7
3 6 7

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;
}
```



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

quote fancy