

Standard Template Library

Mattox Beckman

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF COMPUTER SCIENCE

Spring 2023

Objectives

- ▶ Use the Standard Template Library (STL) built-in datastructures to solve problems
 - ▶ Arrays / Vectors
 - ▶ Stacks and Queues
 - ▶ Sets and Maps

C-Style Arrays

- ▶ A C-style array is simply a block of contiguous memory.
- ▶ First element is always 0.
- ▶ Note the < in the for loop. If you put <= you will certainly have trouble!
 - ▶ If you get a “runtime error” from the judge, check for that.
- ▶ Access an element by index with brackets: $\mathcal{O}(1)$ time. Very fast!
 - ▶ If you have to “look for” an element, it’s $\mathcal{O}(n)$ time. Use with caution.

```
1  int arr[100];
2  int i;
3
4  for(i=0; i<100; ++i) {
5      arr[i] = i * 10;
6  }
```

C-Style Arrays, initialization

- ▶ You can initialize arrays inline in C if you need to.
- ▶ Note that uninitialized items are **undefined**!
 - ▶ You don't have to initialize right away, but you have to before you use it!

```
1 int foo[10] = {8,6,7,5,3,0,9};  
2 char suits[4] = "SHCD"; // Spades, Hearts, Clubs, Diamonds
```

Vectors

- ▶ Vectors in C++ are awesome. Use them unless you have good reason not to.
 - ▶ They can grow dynamically! No need to determine the proper size in advance.
 - ▶ Many iterators to provide traversals.
 - ▶ Reasonable default initialization.
 - ▶ Use `push_back` to insert an element at the end.
 - ▶ Inserting at the beginning is slow! Don't do it!

```
1  vector<int> foo;  
2  
3  for(int i=0; i<N; ++i) {  
4      cin >> data;  
5      foo.push_back(data);  
6  }
```

Vector initializations

- ▶ The constructor can initialize the vector for you.
 - ▶ One argument n : n instances of the default.
 - ▶ Two arguments n and x : n copies of x .

```
1 vector<int> foo(100);  
2 vector<int> foo(500,123);
```

- ▶ You can use `foo.reserve(1000)` to pre-allocate space.

Looping

► C Style

```
1 int sum=0;
2 for(i = 0; i<foo.size(); ++i)
3     sum += foo[i];
```

► Iterator Style

```
int sum=0;
for(auto i = foo.begin(); i != foo.end(); ++i)
    sum += *i;
```

► Reverse Iterator Style

```
1 int sum=0;
2 for(auto i = foo.rbegin(); i != foo.rend(); ++i)
3     sum += *i;
```

Style

- ▶ Certain types come up a lot, so some standard typedefs have evolved:

```
1 typedef vector<int> vi;  
2 typedef vector<vi> vvi;
```


Pairs

- It is often convenient to define tuples as well.

```
1 pair<int,int> coord;  
2  
3 coord.first = 10;  
4 coord.second = 999;
```

- We have standard typedefs for them too.

```
1 typedef pair<int,int> ii;  
2 typedef vector<ii> vii;
```

Stacks

- ▶ I think you know about these....
- ▶ Stacks have three operations:
 - ▶ `push(x)` — add `x` to the top of the stack: $\mathcal{O}(1)$
 - ▶ `pop()` — remove the top element from the stack. (Some implementations will also return the element.) $\mathcal{O}(1)$
 - ▶ `top()` — Returns the top element. $\mathcal{O}(1)$

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  int main() {
4      stack<int> s;
5      s.push(10); s.push(20); s.push(30);
6      while (! s.empty()) {
7          cout << s.top() << endl;
8          s.pop();
9      }
10 } // outputs: 30, 20, 10
```

Stack Use Case

- ▶ Common use-cases: do parens match up?

```
1  stack<int> s;  
2  char data;  
3  
4  while (cin >> data) {  
5      if (data == '(')  
6          s.push(1);  
7      else  
8          s.pop(); // check if empty first though!  
9  }
```

- ▶ Also useful in Depth First Search, cycle detection in graphs.
- ▶ A vector has `push_back`, and can access all members.

Queues

- ▶ Queues have three operations:
- ▶ `push(x)` — add `x` to the back of the queue: $\mathcal{O}(1)$ Traditionally called enqueue.
- ▶ `pop()` — remove the first element from the queue. (Some implementations will also return the element.) $\mathcal{O}(1)$ Traditionally called dequeue.
- ▶ `front()` — Returns the top element. $\mathcal{O}(1)$

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  int main() {
4      queue<int> q;
5      q.push(10); q.push(20); q.push(30);
6      while (!q.empty()) {
7          cout << q.front() << endl;
8          q.pop();
9      }
10 }
```

// outputs: 10, 20, 30

Queue Use Cases

- ▶ You will see these a **lot**.
 - ▶ Many graph algorithms use queues.
 - ▶ Breadth first search
 - ▶ Bipartite graph check
 - ▶ Vectors are not as good a replacement for these.

Motivation

- ▶ Arrays are fun, but what's with all the integers?
 - ▶ Hashmaps, also called *dictionaries*, allow you to look up a value by supplying a key.
 - ▶ E.g., name / phone number, word / definition
- ▶ Hash maps can find *any object* we want quickly.
- ▶ Sets are like hash maps but we don't care about the value part.
- ▶ These, with arrays, are easily the most important data-structure you can know.

Operations

We will show these operations for C++ and PYTHON

- ▶ *Declaring or Creating* the map.
- ▶ *Insert* a key-value pair into the map
- ▶ *Lookup* a value given a key
- ▶ *Check* if a key is in the map
- ▶ *Query* the size
- ▶ *Iterate* over the keys or the values
- ▶ *remove* a key from the map

Creating and Inserting

- ▶ To create these in C++, you will use the map STL class.
 - ▶ You will need to provide the key and the value as templates.
- ▶ Insertion has two forms:
 - ▶ "array like" insertion
 - ▶ "pair" insertion using insert

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int main() {  
    map<string,int> phonebook;  
    phonebook["Jenni"] = 8675309;  
    phonebook["emergency"] = 911;  
    phonebook.insert({"Empire",5882300});  
}
```


In-line initialization

- ▶ You can also initialize it at compile-time, but this is a bit rare in CP.

```
map<string,int> phonebook;  
phonebook = {{ "Jenni", 8675390 },  
              { "emergency", 911 },  
              { "Empire", 5882300 } };
```

Lookup

To lookup a specific value, you also have options:

- ▶ Use array syntax if you know the value is there.
 - ▶ *It will create the key if it doesn't already exist!*

```
cout << phonebook["Jenni"] << " and "  
      << phonebook["H"] << endl;
```

Returns 8675309 and 0.

Finding Keys

- ▶ To check if the key is in the container first, use `contains`

```
if (phonebook.contains("H"))  
    cout << "H is " << phonebook["H"] << endl;
```

- ▶ Finding a specific value is not supported. Program it yourself!

Size

- ▶ To get the number of pairs, use `size()`.
- ▶ To check if it's empty, use `empty()`

```
if (phonebook.empty())  
    cout << "We don't know anyone." << endl;  
else  
    cout << "There are " << phonebook.size()  
        << " entries." << endl;
```

Iteration

- ▶ To loop over all the keys, we have iterators.
- ▶ Note that the order of the keys is arbitrary!
- ▶ Also note that the iterator return pairs!

```
for(auto it = phonebook.begin();  
    it != phonebook.end();  
    ++it)  
    cout << it->first << " has phone number "  
        << it->second << endl;
```

Sets

- ▶ Use `unordered_set` for fast set operations.
- ▶ Use `set` if you want to retrieve the elements in a sorted order.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    unordered<string> people;
    phonebook.insert("Jenni");
    phonebook.insert("emergency");
    phonebook.insert("Empire");
}
```

Creating and Inserting

- ▶ To create in PYTHON, you can initialize an empty version or prepopulate.

```
phonebook = {}
```

```
phonebook["Jenni"] = 8675309  
phonebook["emergency"] = 911  
phonebook["Empire"] = 5882300
```

Also

```
phonebook = {"Jenni":8675390,  
             "emergency": 911,  
             "Empire": 5882300}
```

Lookup and finding keys

To lookup a specific value, you also have options:

- ▶ Use array syntax if you know the value is there.
 - ▶ *It will raise an exception if the key doesn't already exist!*

```
if "H" in phonebook:  
    print(f"{phonebook['Jenni']} and {phonebook['H']}")  
else:  
    print(f"{phonebook['Jenni']}")
```


Finding Values

- ▶ Unlike C++, you can get the values in a dictionary easily:

```
for i in phonebook.values():  
    print(phonebook[i])
```

Size

- ▶ To get the number of pairs, use `len()`.

```
print(f"There are {len(phonebook)} entries.")
```

Iteration

- ▶ To loop over all the keys, we have iterators.
- ▶ Note that the order of the keys is arbitrary!

```
for k in phonebook:  
    print(k)
```

Sets

- ▶ For sets you have to call the `set()` function to start.
- ▶ Use member function `add()` to insert.
- ▶ We also have nice utilities like `intersection()`, `difference()`, etc.

Details

- ▶ In C++, sets are not hashmaps, they typically use red-black trees.
 - ▶ So, $\mathcal{O}(\log_2 n)$ access time.
- ▶ In PYTHON it uses open addressed hashing with random probing for collision resolving.