CAS CS200 LECTURE 1

C++ STL AND SYNTAX

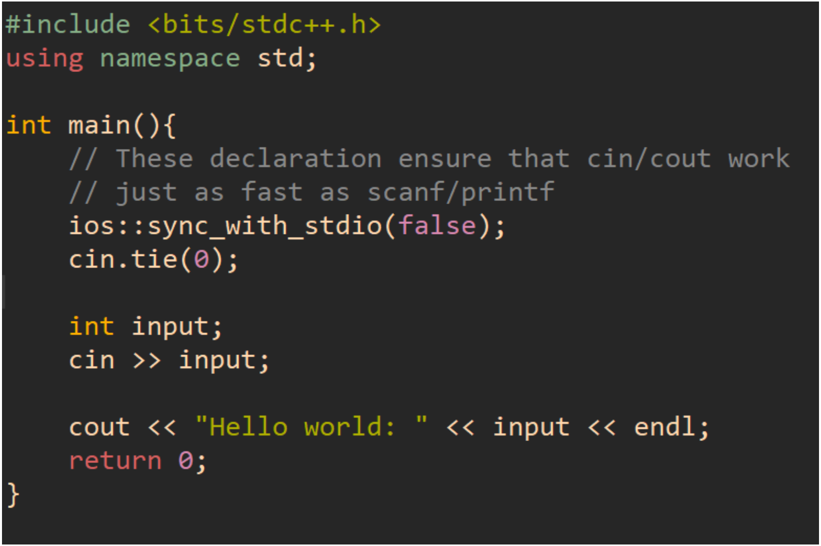
September 19, 2022

C++ in 50 minutes

Objectives

- Understand the basic syntax and I/O in C++
- Useful STL data-structures (containers), their related functions, and basic algorithms

Basic C++ Template



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

int main(){
    // These declaration ensure that cin/cout work
    // just as fast as scanf/printf
    ios::sync_with_stdio(false);
    cin.tie(0);

    int input;
    cin >> input;

    cout << "Hello world: " << input << endl;
    return 0;
}
```

Conventions for "Problem Solving"

General:

- `bits/stdc++` imports all available STL functions
- `using namespace std` saves you time from typing `std::` everytime

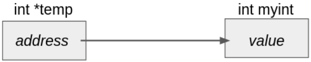
Data Types (`char, int, bool, double, ...`):

- use `std::string` instead of `char[]`
- optionally substitute `int` with `long long` to avoid overflow
- `auto` keyword

Pointers

C++ uses explicit pointers. Declare with asterisk.

- A pointer stores the address of a piece of data in its own data field.
  - Example: a "pointer" `*temp` points to an integer `myint`.



```
temp = &myint;
*temp = 75;
```

- We can get the address with `&`
- Use pointers when:
  - passing data you want to change to a function
  - have a lot of values you want to hold in one data structure

Array

Arrays hold pre-specified amounts of data elements.

- You "can" initialize an array in global scope with a large number based on the known input upperbound. (bad practice for SWE, but easier for problem solving.)
- For fast initialization, use `memset` to fill the array with 0 or -1.

```
int A[1001];
memset(A, -1, sizeof A);
```

- To specify the array content:

```
int A[] = {1,2,3};
```

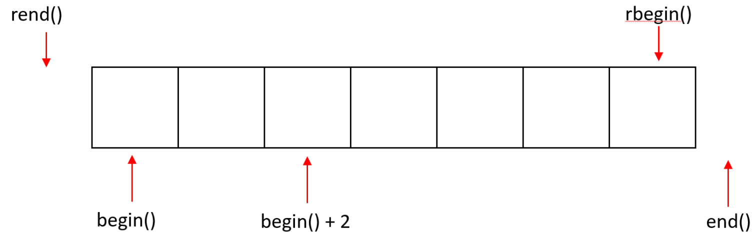
Standard Template Library (STL)

Standard Template Library

- STL contains many useful containers and algorithms.
- Our favorite container: `std::vector⟨T⟩`
  - `map, unordered_map, set, multiset, stack, queue, priority_queue...`
- Our favorite algorithm: `std::sort`
  - `upper_bound, lower_bound, ...`

Iterator

- pointer for STL containers

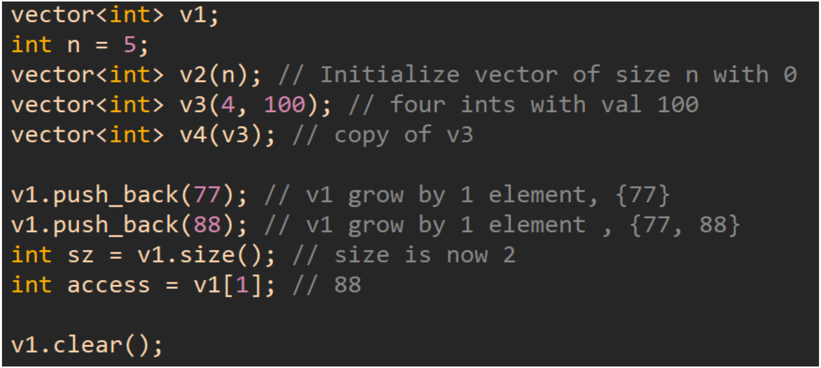


- use `auto` keyword to store them easily
  For example, container v:

```
auto vptr = v.begin()
```

vector

STL vectors are **dynamic**, a predefined size is **optional**.

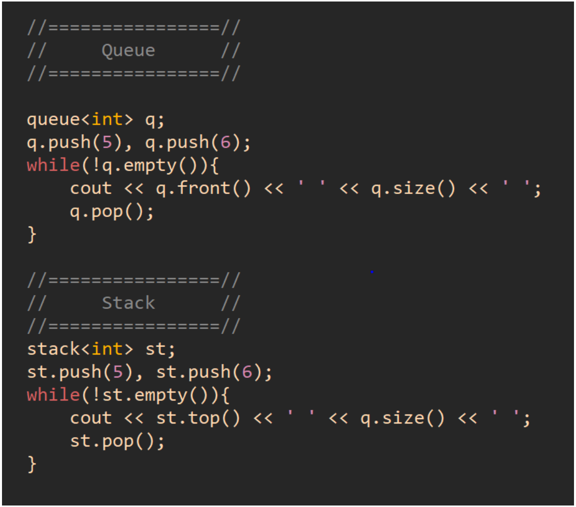


```
vector<int> v1;
int n = 5;
vector<int> v2(n); // Initialize vector of size n with 0
vector<int> v3(4, 100); // four ints with val 100
vector<int> v4(v3); // copy of v3

v1.push_back(77); // v1 grow by 1 element, {77}
v1.push_back(88); // v1 grow by 1 element , {77, 88}
int sz = v1.size(); // size is now 2
int access = v1[1]; // 88

v1.clear();
```

queue/stack



```
//================//
//     Queue      //
//================//

queue<int> q;
q.push(5), q.push(6);
while(!q.empty()){
    cout << q.front() << ' ' << q.size() << ' ';
    q.pop();
}

//================//
//     Stack      //
//================//
stack<int> st;
st.push(5), st.push(6);
while(!st.empty()){
    cout << st.top() << ' ' << q.size() << ' ';
    st.pop();
}
```

Queue : 5 2 6 1 **Stack** : 6 2 5 1

map/unordered_map

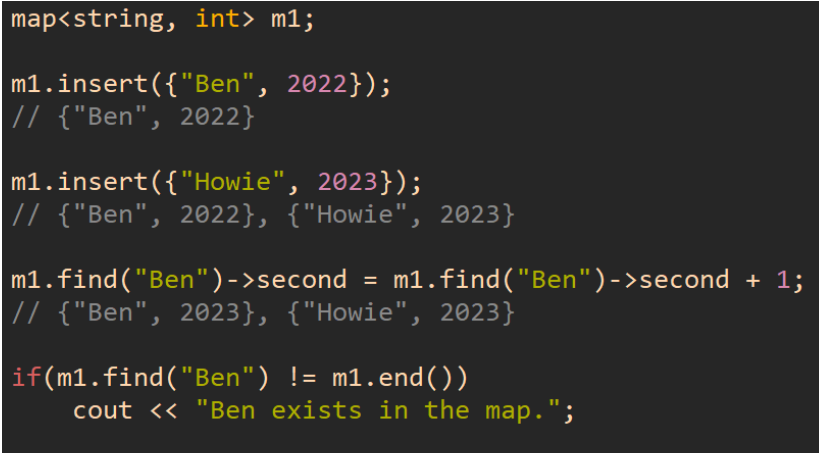
`map` (Implemented as BST)

- Use when keys need to be ordered, traversal is required
- has iterator (i.e `m.begin()`, `m.end()`)
- Search time: `O(logn)`
- Insertion / Deletion `O(logn) + "self-balance" overhead`

`unordered_map` (Implemented as HashMap)

- Use when keeping count, single element access..
- Search time: `O(n)` worst case but `O(1)` on average.
- Insertion / Deletion same as search

map vs unordered_map



```
map<string, int> m1;

m1.insert({"Ben", 2022});
// {"Ben", 2022}

m1.insert({"Howie", 2023});
// {"Ben", 2022}, {"Howie", 2023}

m1.find("Ben")->second = m1.find("Ben")->second + 1;
// {"Ben", 2023}, {"Howie", 2023}

if(m1.find("Ben") != m1.end())
    cout << "Ben exists in the map.";
```

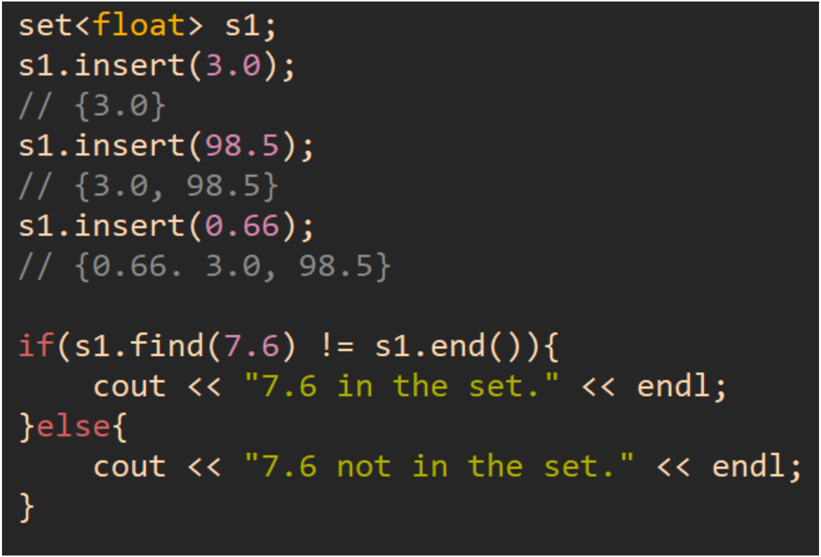
set/multiset

Stores ordered, immutable set of data

- Search time: `O(logn)`
- Insertion/Deletion: `O(logn)` worst case but `O(1)` on average.

Only difference between `set` and `multiset` is the elements are unique/can be duplicate.

set vs multiset

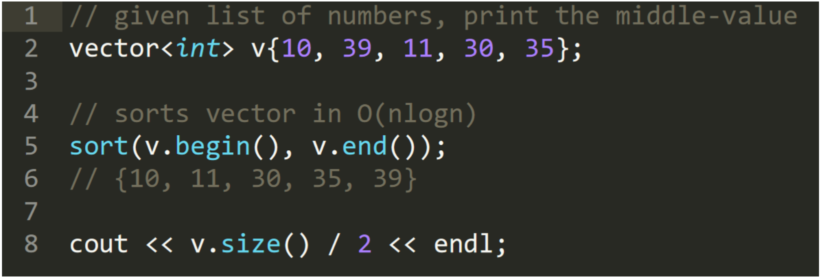


```
set<float> s1;
s1.insert(3.0);
// {3.0}
s1.insert(98.5);
// {3.0, 98.5}
s1.insert(0.66);
// {0.66. 3.0, 98.5}

if(s1.find(7.6) != s1.end()){
    cout << "7.6 in the set." << endl;
}else{
    cout << "7.6 not in the set." << endl;
}
```

sort

Sometimes, it's useful to sort before interaction with a container.



```cpp
// given list of numbers, print the middle-value
vector<int> v{10, 39, 11, 30, 35};

// sorts vector in O(nlogn)
sort(v.begin(), v.end());
// {10, 11, 30, 35, 39}

cout << v.size() / 2 << endl;
```

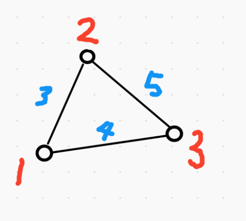
Struct

Classes are secured but have overhead. Use `struct` for custom data.

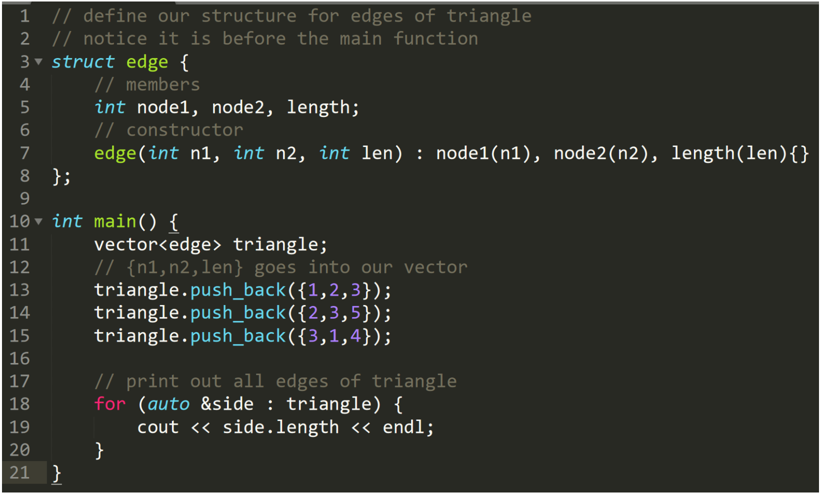
- We can define a `struct` with the following template:

```
struct name{
      int data;
      name(int n):data(n){}
};
```

Let's put what we learned together by creating a data structure to represent this triangle.



Struct



```cpp
// define our structure for edges of triangle
// notice it is before the main function
struct edge {
    // members
    int node1, node2, length;
    // constructor
    edge(int n1, int n2, int len) : node1(n1), node2(n2), length(len){}
};

int main() {
    vector<edge> triangle;
    // {n1,n2,len} goes into our vector
    triangle.push_back({1,2,3});
    triangle.push_back({2,3,5});
    triangle.push_back({3,1,4});

    // print out all edges of triangle
    for (auto &side : triangle) {
        cout << side.length << endl;
    }
}
```

Macros

A common trick people use to take shortcuts is defining macros.

- **Data Types**:

```
typedef long long ll
```

- **Data Structures**:

```
typedef vector<int> vii
typedef pair<int,int> pii
```

- **Functions**:

```
#define rep(i,a,b) for(int i = a; i < (b) ; i++)
#define all(x) begin(x), end(x)
#define pb push_back
```

Trade-off: Less Clarity