Introduction to C++

Xiang Li, 12/30/2024

Preface:

This is a tech talk at the 2024 Winter Hackathon. The talk is intended for beginners who are about to enjoy the Hackathon and provides *only* necessary tools to get started with C++ programming here.

Prerequisite: Basic knowledge of any programming language is helpful.

Contents:

- Hello World in C++
- Data Types, Execution Flow, Functions
- Complexity
- STL Part 1: Array, String, Vector, Stack, Queue, and Iterating Them
- STL Part 2: Priority Queue, Set, Map, Unordered Map
- STL Algorithms: find, reverse, unique, sort, lower_bound, upper_bound
- Tree, Binary Tree

Hello World in C++

```cpp
// "import" the lib to use std::cout
#include <iostream>

// entrypoint of the program, must be exactly like this
int main() {

    // you declare variables like this. It works inside the "scope"
    int a;

    // you can also assign values to variables when declaring
    int b = 1;

    // std::cin is used to read input
    std::cin >> a;

    // std::cout is used to print the result
    // std::endl is used to end the line (prints "\n")
    std::cout << "Hello, World! a+b is " << a+b << std::endl;

    // return 0 to indicate the system your program successfully finished
    return 0;
}
```

Data Types, Execution Flow, Functions

Data Types

The most frequently data types you shall use include:

- `int` and its variants `int64_t`, `uint64_t` (and `bool`)
- `double` and `float`
- `char` and `std::string`
- `struct` that parses some of above data types

- `char` and `std::string`

```cpp
#include <string>
// char example
char c = 'A';  // single character with single quotes

// std::string example
std::string str1 = "Hello, World!";  // a sequence of characters with double quotes

// concatenating strings
std::string str2 = "Hello";
std::string str3 = str2 + ", World!";

// accessing characters in a string
char firstChar = str3[0];

// modifying characters in a string
str3[0] = 'h';
std::cout << str3 << std::endl;  // prints: hello, World!
```

- `struct` that parses some of above data types:

```cpp
struct Point {
    int x, y;
    double z;
};

Point p;  // use `Point` to declare a variable `p`
p.x = 1;  // use `.` to access the member of a struct
p.y = 2;
p.z = 3.0;
```

Execution Flow

- C++ executes from the start of `main` function to the end.
- `if/else` is used to creat branches.

```
if (a > 0) {
    // do something
} else if (a > -3) {
    // do something else
} else {
    // do something else
}
```

`else` is not compulsory. But be careful if you have `return` in the `if` block.

- `for` loop and `while` loop are frequently used to iterate.

```cpp
int j = 0;
while (j < 10) {
    // do something 10 times
    j++;  // add one to j, the counter
}
// j is still accessible here

for (int i = 1; i <= 100; i++) {
    if (i % 15 == 0) {
        std::cout << "FizzBuzz" << std::endl;
        continue;  // skip the rest of the loop (same in `while` loop)
    } else if (i == 70) {
        break;  // exit the most inner loop (same in `while` loop)
    }
    std::cout << i << std::endl;
}
// warning: i is not accessible here, it is out of scope
```

Functions

- Functions are used to encapsulate a block of code.
- You likely want a function if you want to
    - reuse the code
    - make the code more readable
    - make the code more maintainable

```
// a function looks like:
//
// <return type> <function name>(<parameters>) {
//     // do something (function body)
//     return <return value>;
// }
int add(int a, int b) {
    return a + b;
}
// return type `void` means the function does not return anything
```

Copy vs. Reference

- By default, C++ passes arguments to function by value.
- This means **modification inside will not affect the original one.**

```cpp
int add_one(int a) {
    a = a + 1;
    return a;
}

int main() {
    int a = 1;
    int b = add_one(a);  // you don't expect `a` gets changed, so `a` stays 1
    return 0;
}
```

However, you will likely need to change the original value inside the function. You can use **reference** to do this.

```
int add_one(int &a) {  // reference mark `&` is used
    a += 1;   // equivalent to a = a + 1;
    return a; // actually you don't need to return `a` here
}

int main() {
    int a = 1;
    int b = add_one(a);  // you expect `a` gets changed, so `a` becomes 2
    return 0;
}
```

Containers in C++

- There are many containers in C++ to store data. Access them by including the corresponding header file.
- The most frequently used containers include: (some of them require C++11)
  - `std::array` : fixed-size array (the only one that is **fixed-size**)
  - `std::vector` : dynamic array
  - `std::stack` : LIFO stack
  - `std::queue` : FIFO queue
  - `std::priority_queue` : priority queue, heap
  - `std::set` : ordered set
  - `std::map` : ordered map
  - `std::unordered_map` : unordered map

You cannot mix different types in a single container.

How to Initialize Containers

- default initializer for Them

```cpp
// default initialize with values initialized as well
std::array<int, 5> arr;  // {0, 0, 0, 0, 0}

// initialize with empty container
std::vector<int> vec;  // {}
std::stack<int> stk;
std::queue<int> que;
std::priority_queue<int> pq;
std::set<int> s;
std::map<int, int> m;
std::unordered_map<int, int> um;
```

- `std::array` and `std::vector` can be initialized like this:

```cpp
#include <array>
#include <vector>

// array requires known size at compile time
std::array<int, 5> arr = {1, 2, 3};  // {1, 2, 3, 0, 0}
std::vector<int> vec = {1, 2, 3, 4, 5};
std::vector<int> vec2(5, 0);  // {0, 0, 0, 0, 0}

// vector of vectors
std::vector<std::vector<int>> vec2d = { {1, 2}, {3, 4, 5}, {6} };
```

```cpp
// append elements into containers
vec.push_back(6);   // {1, 2, 3, 4, 5, 6}
stk.push(1);        // stack {1}
que.push(2);        // queue {2}
pq.push(3);         // priority_queue {3}
s.insert(4);        // set {4}
m[5] = 5;           // map {5: 5}
um[6] = 6;          // unordered_map {6: 6}

// get the size of any container
int size = vec.size();  // also works for other containers, std::string
```

```cpp
// iterate through vectors/arrays in the traditional way
for (int i = 0; i < vec.size(); i++) {
    std::cout << vec[i] << std::endl;
}

// or range-based for loop (C++11)
for (int x : vec) {
    std::cout << x << std::endl;
}

// or use iterators for more containers
auto vit = vec.begin();
while (vit != vec.end()) {
    std::cout << *vit << std::endl;
    vit++;
}
```

```
// iterate through associative containers with iterators
for (auto it = m.begin(); it != m.end(); it++) {
    std::cout << it->first << ": " << it->second << std::endl;
}

// or range-based for loop (C++11)
for (auto [key, value] : m) {
    std::cout << key << ": " << value << std::endl;
}
```

The iteration also works for `std::string`.

- count a given value in a container

```cpp
std::vector<int> vec = {1, 2, 3, 3, 4, 5};

int count = std::count(vec.begin(), vec.end(), 3);
```

STL Algorithms

- STL provides many algorithms to operate on containers.
- We introduce some for `std::vector` (also work for `std::array`).
- Some of them also work for other containers.

`std::find`

- Simple but better for readability and maintainability (and shorter than writing it yourself).

```cpp
#include <algorithm>
std::vector<int> data {1, 2, 3, 4, 5};

int val = 3;

// type of result is also an iterator
auto result = std::find(data.begin(), data.end(), val);

if (result != data.end()) {
    std::cout << "Found " << val << " at pos " << std::distance(data.begin(), result) << std::endl;
} else {
    std::cout << val << " not found in the data." << std::endl;
}
```

`std::reverse`

- Reverse the order of elements in a vector, array.

```
std::vector<int> data {1, 2, 3, 4, 5};

std::reverse(data.begin(), data.end());
```

`std::unique`

- Move the unique elements to the front of the container.
- and you can erase the unspecified values.

```cpp
std::vector<int> data {1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5};

auto last = std::unique(data.begin(), data.end()); // logical end of the unique elements

data.erase(last, data.end()); // Erasing the unspecified values
```

`std::sort`

- Sort the elements in the container in $O(n \log n)$ time (average and worst).
- Use `std::stable_sort` if you want to keep the relative order of equal elements.

```cpp
std::vector<int> data {5, 3, 1, 4, 2};

// Sort in ascending order
std::sort(data.begin(), data.end());
```

`std::lower_bound` and `std::upper_bound`

- Find the first element index v in sorted array a of x such that
  - lower: $a[v] \geq x$; upper: $a[v] > x$.

```cpp
std::vector<int> data {1, 2, 4, 4, 5, 7, 8};

// Sort the vector (required for std::lower_bound and std::upper_bound)
std::sort(data.begin(), data.end());

// Find lower bound of 4
auto lower = std::lower_bound(data.begin(), data.end(), 4);
std::cout << "Lower bound of 4 is at index: " << (lower - data.begin()) << std::endl;

// Find upper bound of 4
auto upper = std::upper_bound(data.begin(), data.end(), 4);
std::cout << "Upper bound of 4 is at index: " << (upper - data.begin()) << std::endl;


// Lower bound of 4 is at index: 2
// Upper bound of 4 is at index: 4
```

Note: `std::set` and `std::map` have built-in `lower_bound` and `upper_bound` methods.

Tree

- A tree is a data structure that consists of nodes in a parent/child relationship.
- To define a tree, we first need to learn what is pointer.

Pointer

```cpp
int a = 1;
int *p = &a;  // p is a pointer to a, &a is the address of a

// pointer can be dereferenced to get the value
int b = *p;  // b is 1

// pointer might be null
int *q = nullptr;  // q is a null pointer
// you cannot dereference a null pointer
// int c = *q;  // this will cause a runtime error

// pointer p, q have type "int *" (a pointer that points to an int)

// pointer can point to pointer (and anything)
int **pp = &p;  // pp is a pointer to p, it has type "int **"
```

- Then we can define a node in a tree.

```cpp
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// use new to create a new node
TreeNode *root = new TreeNode(1);
root->left = new TreeNode(2);
root->right = new TreeNode(3);

// you should delete the nodes when you don't need them (note the order)
// delete root->left;
// delete root->right;
// delete root;
//
// if you don't delete them, it will cause memory leak
// but fine for this hackathon
```

Binary Tree in Array

- A binary tree can be stored in an array. (fixed number of children)

```cpp
//        1
//       / \
//      2   3
//     / \ / \
//    4  5 6  7

std::vector<int> tree = {-1, 1, 2, 3, 4, 5, 6, 7};
```

- index 0 not used (or set to -1), index 1 is the root.
- index i's left child is at $2i$, right child is at $2i+1$.

Thank you for your attention!