CS 4414: Recitation 2

Sagar Jha

Today: More C++ (Types, Containers)

- We will talk about C++ types, std::vector and std::map

- Basic C++ philosophy
  - RAII: discussed in the last recitation, will see more of it in the future
  - C++ prioritizes performance: more compile time optimizations, less runtime checks
  - Gives programmers control over performance, places a lot of faith on them to write correct code
  - C++ aims to be backward compatible with C and older versions of C++. Many obscure, outdated features exist in C++.

Variables

- A C++ variable has a name, a type, a value and an address in memory

```
int x = 5;
```

- Can obtain the address (represented in hex) with the & operator

```
std::cout << &x << std::endl;
```

```
~ $
~ $ ./my_program
0x7ffd55b5daa4
~ $
```

Types

- Primitive data-types: bool, char, int, float, double…
- Size of a type is implementation defined, use sizeof to find the size

```
std::cout << sizeof(int) << std::endl;
```

- User defined types: struct, class…

```
class MyClass {

  int myVar;

public:
  void myFun() {}

};
```

```
MyClass my_object;
```

Pointer and array type

- A pointer stores the memory address of a variable. (correction: int* p = &x)

```
int* p = x;  // p "points" to x
```

- The variable can be accessed by dereferencing the pointer. Beware of null-dereferencing!

```
std::cout << p << std::endl;  // prints 0x7ffc46cd8054

std::cout << *p << std::endl;  // prints 5
```

- Size of a pointer is the size of a memory address – 4 Bytes on a 32-bit machine, 8 Bytes on 64-bit (1 Byte = 8 bit)

Pointer and array type

- Pointer arithmetic: Adding 1 to a pointer returns the address of the next variable

```
std::cout << p << " " << p + 1 << std::endl;  // print 0x7ffd79f5034c 0x7ffd79f50350
```

- Native arrays can be seen as pointers

```
int arr[5];

x = arr[2];  // or *(arr + 2)
```

- Char** - pointer to a char*, represents an array of strings

Bool and char type, auto keyword

- A bool is a single bit. Its value is 0 or 1 (false or true)
- A char is 1 Byte on most machines, can take values from 0 to 255

```
if (ch - '0' >= 0 && ch - '0' <= 9) {}
```

- Beware of implicit conversions! (correction my_ptr != nullptr)

```
// if (my_int) {}  // equivalent to if (my_int != 0)
// if (my_ptr) {}  // equivalent to if (my_ptr == nullptr)
```

Bool and char type, auto keyword

- Compiler infers type of variable defined with the auto keyword

```
int max (int x, int y);

auto m = max(x, y);  // m is an int, the return type of max
```

Class

- Class initializer list in the constructor, this points to the object

```
wc::wordCounter::wordCounter(const std::string& dir, uint32_t num_threads)
        : dir(dir),
          num_threads(num_threads) {
}
```

```
wc::wordCounter word_counter_one("/home/sagar/Documents", 4);

wc::wordCounter word_counter_two{"/home/sagar/Documents", 4};

wc::wordCounter word_counter_three = wc::wordCounter("/home/sagar/Documents", 4);

wc::wordCounter word_counter_fourth = {"/home/sagar/Documents", 4};
```

- Don’t use new, that returns a pointer to the object!

Type qualifiers (const, volatile)

- A const variable cannot change state after declaration

```
std::cin >> x;

const int y = x;  // y's value cannot change
```

```
~ $
~ $ g++ -std=c++17 my_program.cpp -o my_program
my_program.cpp: In function ‘int main(int, char**)’:
my_program.cpp:101:16: error: passing ‘const MyClass’ as ‘this’ argument disca
rds qualifiers [-fpermissive]
  101 |    my_obj.print();
      |               ^
my_program.cpp:16:8: note:   in call to ‘void MyClass::print()’
   16 |    void print() {
      |         ^~~~~
~ $
```

```
~ $
~ $ g++ -std=c++17 my_program.cpp -o my_program
my_program.cpp: In member function ‘void MyClass::print() const’:
my_program.cpp:18:13: error: assignment of member ‘MyClass::myVar’ in read-onl
y object
   18 |       myVar = 0;
      |       ~~~~~~^~~
~ $
```

Type qualifiers (const, volatile)

- Const vs. constexpr – constexpr's value is known at compile-time

```
main.cpp:23:12: warning: ISO C++ forbids variable length array ‘args’ [-Wvla]
   23 |     string args[argc];
```

Plain Old Data (POD)

- Why must array size be constant at compile time?
- A POD type is a class or struct without pointers, constructors/destructors and virtual member functions
- Why is a POD type useful?
  - All the struct’s data is stored in contiguous memory. This enables some optimizations and one can reliably copy the struct by copying the memory contents
- A struct can contain native arrays and still be POD

Source: https://stackoverflow.com/questions/146452/what-are-pod-types-in-c

When to use pointers

- Prefer objects always over pointers, std::vector or std::array over native arrays
- If an object must be shared across multiple classes, prefer smart pointers (std::unique_ptr<T>, std::shared_ptr<T>)
- Read: https://stackoverflow.com/questions/22146094/why-should-i-use-a-pointer-rather-than-the-object-itself

Standard Template Library

- Collection of classes and functions for general-purpose use
- Provides container types (list, vector, map), pair, tuple, string, thread and many other functionalities
- Available in the std namespace

std::vector<T> – Most important C++ container

- A dynamic array – Can be resized as required, initial size 0 if not specified
- Memory representation: elements are stored contiguously in memory
- Provides O(1) random access with [] or std::vector<T>::at, no bounds checking with []
- std::vector<T>::push_back(const T& value) – append to the end of the vector. Similarly pop_back. Amortized O(1) complexity
- Size vs. capacity. Do not confuse with sizeof.
- Memory reallocation on resizing or push_back, prefer constructing vectors with the total size and then filling in elements
- O(n) complexity for insertion and removal at a random position in the vector

std::vector<T> vs. std::list<T>

- A C++ list is a collection of elements at non-contiguous locations in memory, linked using pointers
- Provides O(1) insertion and deletion from any location of the list, but O(n) complexity for random access

std::vector<T> vs. std::list<T>

```
std::vector<fs::path> utils::find_all_files(
        const fs::path& dir, std::function<bool(const std::string&)> pred) {
    std::list<fs::path> files_to_sweep;
    // iterate recursively to find all files that satisfy pred
    for(auto& entry : fs::recursive_directory_iterator(dir)) {
        if(entry.is_regular_file()) {
            fs::path cur_file = entry.path();
            std::string type(cur_file.extension());
            if(pred(type)) {
                files_to_sweep.push_back(std::move(cur_file));
            }
        }
    }
    return std::vector<fs::path>(
            std::make_move_iterator(files_to_sweep.begin()),
            std::make_move_iterator(files_to_sweep.end()));
}
```

std::map<K, V> - Second most important container

- Maps keys to values
- std::map<K,V>::at vs []
- Use a map when you need to access elements by key, a vector when you need to access by position
- Implementation using trees, O(log n) complexity for insert, remove, erase, search
- std::unordered_map<K, V> - hash-based map, O(1) but unpredictable complexity. Prefer std::map unless there is a specific reason
- std::insert ignores if key is already present!

We often need to convert between containers

```
void wc::wordCounter::display() {
    // to print in sorted value order (frequency), convert the map to a vector of pairs and then sort the vector
    using pair_t = std::pair<std::string, uint64_t>;
    std::vector<pair_t> freq_vec(freq.size());
    uint32_t index = 0;
    for(auto [word, cnt] : freq) {
        freq_vec[index++] = {word, cnt};
    }
    std::sort(freq_vec.begin(), freq_vec.end(), [](const pair_t& p1, const pair_t& p2) {
        // decreasing order of frequency. Break ties alphabetically
        return p1.second > p2.second || (p1.second == p2.second && p1.first < p2.first);
    });

    for(auto [word, cnt] : freq_vec) {
        std::cout << word << ": " << cnt << std::endl;
    }
}
```