CS4414 Recitation 2
C++ Types and Containers

09/03/2021
Sagar Jha
C++ is strongly typed!

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

```cpp
int x = 5;
```

- **Name**
- **Type**
- **Initial Value**
C++ is strongly typed!

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

```cpp
int x = 5; // declaration + initialization
```

• Later, you can use variable `x` in expressions such as,

```cpp
int y = x + 1; // initialization of y using x
x = 7; // reassignment
```
Address and initial value

• Can obtain the **address** (represented in hex) with the `&` operator

```cpp
std::cout << &x << std::endl;
// prints 0x7ffd55bd64a4
```

• What happens if you use an **uninitialized** variable?

```cpp
int x;
std::cout << x << std::endl;
```
Address and initial value

• Can obtain the **address** (represented in hex) with the `&` operator

```cpp
std::cout << &x << std::endl;
// prints 0x7ffd55bdac4
```

• What happens if you use an **uninitialized** variable?

```cpp
int x; // undefined value
std::cout << x << std::endl;
// prints 0 on my machine
```
Types

• Primitive data types
  • bool
  • char
  • int
  • float
  • double

• Derived data types
  • pointer
  • array
  • function

• User-defined data types
  • struct
  • class
Primitive data types

- **bool**: Represents two values – **true** and **false**
- **char**: a-z, A-Z, 0-9, special characters such as space, newline etc.
- **int**: Represents integer values
- **unsigned int**: Represents integer values >= 0
- **float, double**: Represents floating point numbers
Each C++ type has a fixed size, but...

• the size is implementation defined in general
• Lots of integer types
  • int, short, unsigned int, long, long long, unsigned long...
  • even more: int8_t, int16_t, int32_t, int64_t,...
• Use sizeof(<type>) to find the size

```cpp
long long int x = 0;
std::cout << sizeof(x) << std::endl;  // prints 8
std::cout << sizeof(long long int) << std::endl; // prints 8
```
Question: What’s the largest value that a 4-byte integer can represent?
Question: What’s the largest value that a 4-byte integer can represent?

• 4 bytes = 32 bits
  A 32-bit datatype can represent \(2^{32}\) distinct values
• A signed 4-byte integer can represent numbers from \(-2^{31}\) (-2, 147, 483, 648) to \(2^{31} – 1\) (2, 147, 483, 647)
• An unsigned 4-byte integer can represent numbers from 0 to \(2^{32} – 1\) (4, 294, 967, 295)
• **Tip**: Use fixed-width integer types defined in **cstdint**. 4-byte integers for normal use (int32_t, uint32_t) and 8-byte integers (int64_t, uint64_t) for representing larger values
Operators

- Arithmetic: $a + b$, $a - b$, $a \times b$, ...
- Logical: $!a$, $a \&\& b$, $a \| b$
- Relational: $a == b$, $a < b$, $a > b$, $a <= b$, ...
- Assignment: $a = b$, $a += b$, $a /= b$, ...
- Increment: $++a$, $--a$, $a++$, $a--$

```java
if (x + y < 7 && !(z > 10)) {
    // do something
}
```
• $x += y$ is equivalent to writing $x = x + y$
• can also use for bools: $b1 |= b2$
More on increment and decrement

• Pre-increment (\(++a\)) and post-increment (\(a++\)) behave differently

```
x = ++y; or
x = y++;  
```

```
x y
2 3
```

```
x y
? ?
```
More on increment and decrement

- Pre-increment (++a) and post-increment (a++) behave differently

![Diagram showing the difference between pre- and post-increment]

- x = ++y;
- x = y++;

Values:
- x = 3
- y = 4
Implicit conversions

- false is 0, true is 1. Any non-zero int is true, int 0 is false

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

- Implicit conversion from char to int (using ASCII codes)

  ```
  isdigit(ch): ch >= 48 && ch <= 57
  ```
Implicit conversions

• false is 0, true is 1. Any non-zero int is true, int 0 is false

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

• Implicit conversion from char to int (using ASCII codes)

```c
isdigit(ch): ch >= 48 && ch <= 57
```

written better as,

```c
isdigit(ch): ch >= '0' && ch <= '9'
```
C++ auto keyword and const qualifier

• Compiler infers type of variable defined with the auto keyword

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

• const keyword before a variable declaration fixes its value to the initial value

```cpp
const double pi = 3.14; // good for readability
```
More in future recitations

POINTERS

CLASSES
Exercise: Explain the error!

```cpp
#include <iostream>

class myClass {
public:
    void print () {
        std::cout << "My integer is: " << myInt << std::endl;
    }

private:
    int myInt = 10;
};

int main() {
    const myClass myObj;
    myObj.print();
}
```

```bash
$ g++ program.cpp -o program
$ g++ program.cpp -o program
program.cpp: In function 'int main()':
program.cpp:16:15: error: passing 'const myClass' as 'this' argument discards qualifiers [-fpermissive]
    16 |     myObj.print();
         ^
    16 |     myObj.print();
    16 |     myObj.print();
    5 |     void print () {
       ^~~~
$ 
```
Exercise: Explain the error!

- print function can potentially change the state of a myClass object, so it cannot be called on a const object
- To assert that print cannot change object state, change it to void print () const { ... }
Follow up: What happens when myInt is incremented in the const print function?

```cpp
~ $ g++ program.cpp -o program
program.cpp: In member function ‘void MyClass::print() const’:
program.cpp:7:5: error: increment of member ‘MyClass::myInt’ in read-only object
  7 |     myInt++;
    | ^~~~~~
~ $
```
Part II : Containers

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> and std::array<T, N>

• T is a template parameter
• std::vector<int> is a vector of integers, std::vector<char> is a vector of characters
• T can be a class or other C++ container. E.g., std::vector<std::vector<int>>, std::vector<std::map<int, std::string>>...
Array – a fundamental datatype

- O(1) access given the index (or position) of the element
- Stores elements contiguously (in continuous memory locations)
- Elements are accessed starting with position 0 (0-based indexing)
- How to access the element at a given position in O(1) time?
The size of an array is constant in C++

• `std::array<int, 10> my_array = {1, 2, 3};` defines an integer array of size 10

• The size must be fixed at compile-time

• Elements accessed using the [] operator. For e.g., `my_array[2]` is 3

• Note: No bounds checking!

• Question: What happens if you do `my_array[20]` with only 3 elements?
std::vector<T> - A dynamic-sized array

• Main problem: How to support inserting elements efficiently?
• Concept of size vs. capacity
std::vector<T> - A dynamic-sized array

• Main problem: How to support inserting elements efficiently?
• Concept of size vs. capacity
• Reallocates elements when capacity is exceeded
Complexity of std::vector<T>::push_back

- Most push_backs will be $O(1)$ (when size < capacity)
- Some will have linear complexity (when the vector is reallocated)
- Amortized $O(1)$ complexity with exponential growth in capacity
- What about the complexity of inserting at a random position in the vector?
Complexity of std::vector<T>::push_back

• Most push_backs will be O(1) (when size < capacity)
• Some will have linear complexity (when the vector is reallocated)
• Amortized O(1) complexity with exponential growth in capacity
• What about the complexity of inserting at a random position in the vector?

std::vector<T>::insert (iterator pos, const T& value)

Must shift elements to the right! Linear complexity
Exercise

• Pick a large N (> 1 million)
• Program A: Creates a vector of N elements and assigns $\text{vec}[i] = i$ for each $i$ in a for-loop
• Program B: Creates an empty vector and calls $\text{vec.push_back}(i)$ $N$ times in a for-loop
• Program C: Creates an empty vector and calls $\text{vec.insert(\text{vec.begin()}, N - i - 1)}$ $N$ times in a for-loop
• Measure the time taken by A, B, and C