C++ London Uni

Initial C++ — session 5

Tristan Brindle

Feedback

- We'd love to hear from you!
- The easiest way is via the *cpplang* channel on Slack — we have our own chatroom, *#cpplondonuni*
- Go to https://cpplang.now.sh/ for an "invitation"

Last week…

- Value semantics
- Pass by value
- Scope
- Basic object lifetimes
- Basic control flow
  - If statements

This week

- Loops
- References and const references
- Pass-by-reference

Last week's homework

1. Write a program that reads in a sequence of `double`s from the user using `std::cin`. Print out the *minimum* and *maximum* values that they entered. Can you do this without storing every entered value?

2. Extend your program so that it also prints out the *mean* of the numbers the user entered (Hint: this time you may want to use a `std::vector` to store the input values to make the calculation easier).

3. Extend your program so that it also prints out the *median* of the numbers the user entered

4. (Harder): Extend the program so that it prints out the *mode* (that is, the value that appears most often) of the input sequence. (Hint: there may be more than one such value.)

Solution to last week's homework

- https://github.com/CPPLondonUni/initial_week4_homework_soln
- See the various branches for solutions to individual exercises

Any questions before we move on?

Loops

- Last week we looked at *if statements*, one of the basic building blocks of programs
- Another of the fundamental building blocks are *loops*
- A *loop* tells the program to keep repeating some action while a *condition* is true
- C++'s loop keywords are very similar to those found in other languages

While loops

- The simplest form of loop uses the `while` keyword
- We write this as

```
while (some_condition) {
    // do_something
}
```

- The program keeps repeating the *body* of the loop until the *condition* is `false`
- If the condition is `false` to start with, we will never enter the loop

Do-while loops

- Similar to a while loop is a do-while loop. This is written

```
do {
    // something
} while (condition);
```

- Unlike a plain while loop, the body of do is executed once *before the condition is checked*

- This means that a do-while loop will execute *at least once*

For loops

- The most common type of loop is the *for loop*
- This has the form

```
for (init ; condition ; increment) {
    // do something
}
```

- First the *init* part is run, then the *condition* is checked
- If the *condition* is `true`, the body of the loop is executed, followed by the *increment* step
- This is repeated until the *condition* is `false`

For loop example

```
int main()
{
    for (int i = 0; i < 10; i++) {
        std::cout << "i is " << i << '\n';
    }
}
```

Range-for loops

- C++ also has range-for loops, which can be used to loop over collections (*ranges*)

- This has the form

```
for (declaration : range) {
    // do something
}
```

- This is extremely useful when dealing with vectors and strings!

- A range-for loop can be used with any type that meets the *range concept* requirements, including all standard library collections

Range-for example

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

    for (int i : vec) {
        std::cout << i << '\n';
    }
}
```

Break and continue

- Within the body of a loop, the `break` keyword can be used to *immediately exit* from the loop

- The `continue` keyword immediately jumps to the *end* of the loop — the condition will be tested and, if appropriate, the loop body will be entered again

- In nested loops, `break` and `continue` only apply to the innermost loop

Exercise 1

- Write a function named `print_squares` which takes a `std::vector<double>` parameter.

- As its name suggests, this function should print out the *square* of each element of the given vector

  1. Using a range-for loop

  2. Using an ordinary for loop

  3. Using a while loop

Solution

```cpp
#include <iostream>
#include <vector>

void print_squares1(std::vector<double> vec)
{
    for (auto d : vec) {
        std::cout << d * d << ' ';
    }
}

void print_squares2(std::vector<double> vec)
{
    for (int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] * vec[i] << ' ';
    }
}

void print_squares3(std::vector<double> vec)
{
    int i = 0;
    while (i < vec.size()) {
        std::cout << vec[i] * vec[i] << ' ';
        ++i;
    }
}
```

Any questions before we move on?

Revision: value semantics

- Unlike many other programming languages, C++ uses *value semantics* by default

- This means (roughly) that *copies of variables are distinct*; changing the value of a copy will not affect the original variable (i.e. copies are “deep”).

Value semantics example

```cpp
int a = 1;

int b = a;
// b has value 1

a = 42;
// a has value 42
// b still has value 1

std::string s = "Hello";
std::string s2 = "World";

s += s2;
// s now has value "HelloWorld"
// s2 still has value "World"
```

Reference semantics

- Value semantics are great: they allow us to *reason locally* about our functions
- However, copying large amounts of data for each function call is potentially very slow
- This is one of the reasons C++ allows us to optionally use *reference semantics* as well
- References are also needed for object-orientated programming, as we'll see later in the course

References in C++

- We can declare a reference to an existing variable by writing an ampersand & after the type name, for example:

```
int i = 0;
int& ref = i;
```

- You always need to initialise a reference!

- You can think of a reference as a *new name* for an existing variable

- **Anything we do to the reference affects the original variable**

References example

```
int i = 0;     // i is an an int
int& ref = i; // ref is a reference to int

ref = 42;  // i is now equal to 42

i = 24; // i is now equal to 24

std::cout << ref << '\n';
// prints 24

int& ref2 = ref;
// ref2 is another reference to i
```

Pass by reference

- As we saw last week, C++ uses *pass-by-value* by default for function calls
- This means that a function will receive a *copy* of the arguments you pass to it
- But we can also use *pass-by-reference*, by declaring that a function parameter has *reference type* (with &)
- This means that the function will operate *directly* on the argument you pass to it, not a copy

Pass-by-reference example

```
void say_hello(std::string& name)
{
    name = "Hello " + name;
}

std::string me = "Tristan";

say_hello(me);

std::cout << me << ‘\n';
// prints Hello Tristan
```

Exercise 2

- In CLion, create a new C++ executable project
- In `main.cpp`, write a function `void increment(int& i)` which adds one to `i`
- In your `main()` function, call `increment()` on a local `int` variable and print the result to see that it modifies the argument you pass to it
- Try removing the & from the definition of `increment()`. Does the code still compile? What happens? Why?
- Try calling `increment(42)`. Does the code still compile? What happens? Why?

Solution

```cpp
#include <cassert>
#include <iostream>

void increment(int& i)
{
    ++i;
}

int main()
{
    int val = 0;

    increment(val);

    assert(val == 1);

    std::cout << val << '\n'; // prints 1
}
```

Exercise 3

- Write a new function

```
swap(std::string& s1, std::string& s2)
```

- In this function, swap the values of `s1` and `s2` — that is, after the function returns, `s1` should contain the old value of `s2`, and `s2` should contain the old value of `s1`

- Use `cout` or `assert()` to show that your function works correctly

Solution

```cpp
#include <string>
#include <cassert>

void swap(std::string& s1, std::string& s2)
{
    auto temp = s1;
    s1 = s2;
    s2 = temp;
}

int main()
{
    std::string s1 = "Hello";
    std::string s2 = "World";

    swap(s1, s2);

    assert(s1 == "World" && s2 == "Hello");

}
```

Homework

1. Write a function `lower_case()` which takes a reference to a `std::string`, and modifies the string so that every character is changed to lower case. For example, the string "MiXeD" should be changed to "mixed".

2. Write a function `headline_case()` which takes a reference to a string. Change the first character of every *word* of the string into a capital letter. For example, the string "this is a headline" should be changed to "This Is A Headline"

3. Write a function `sentence_case()` which changes the first character of every *sentence* into a capital letter. Sentences are separated by a full stop followed by a single space character. For example, the string "this is sentence 1. this is sentence 2." should be changed to "This is sentence 1. This is sentence 2."

4. Write a function `sentence_case2()` which does the same as `sentence_case()`, except that sentences may be separated by an arbitrary number of whitespace characters. For example, the string "sentence 1. \n\t\r sentence 2." should be changed to "Sentence 1. \n\t\r Sentence 2." (HINT: the function `std::isspace()` can be used to check for whitespace characters.)

5. Write a function `sentence_case3()` which does the same as `sentence_case2()`, except that *all* whitespace characters between sentences are replaced with a *single* space character. For example, the string "sentence 1. \n\t\r sentence 2." should be changed to "Sentence 1. Sentence 2."

Online resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)