C++ London
uni

Initial C++ — session 2

Tristan Brindle

Register now for C++ on Sea!

https://cpponsea.uk/

About these sessions

- An introduction to C++
- A mixture of talks, class exercises and homework
- We can’t turn you into an expert (sorry!)
- …but we’ll try to give you enough information to get started

Feedback

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

Last week…

- First introduction to C++
- Deconstructing “Hello world”
- Types in C++
- Declaring and calling functions

This week…

- Commonly used types
- Variable declarations
- `const` and `auto`

Functions revision

- The usual form of a function declaration is

```
return-type function-name(param-type param-name, …)
```

- Every function in C++ returns zero or one value(s)
- If the function does not return a value, then the return type is `void`

Functions revision

- For example, we can define a function which adds two `int`s like so:

```
int add(int a, int b)
{
    return a + b;
}
```

- This defines a function “add” which takes two parameters named *a* and *b* (both of type `int`) and returns a value of type `int`

Functions exercise

- On wandbox.org, write a function `say_hello()` which takes a parameter of type `std::string` called `name`, and returns a string containing that name with “Hello ” in front

- Use this function to print “Hello <your name>” from `main()`, e.g. “Hello Tristan”

- You will need to add `#include <string>` near the top of your program to use `std::string`

Solution

```
#include <iostream>
#include <string>

std::string say_hello(std::string name)
{
   return "Hello " + name;
}

int main()
{
    std::cout << say_hello("Tristan") << '\n';
}
```

Last week's homework

- Read about `std::cin`, the input counterpart to `std::cout`

- Write a program which reads in a string using `std::cin`, and then prints "Hello, <name>", where <name> is the string the user supplied

- Bonus: use `std::vector` to read in a list of strings. When an empty string is entered, print "Hello <name>" for each string that was entered so far, and then exit the program.

Solution (1)

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name;

    std::cin >> name;

    std::cout << "Hello, " << name << '\n';
}
```

Solution (2)

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> names;

    std::string name;

    while (std::cin >> name) {
        names.push_back(name);
    }

    for (auto n : names) {
        std::cout << "Hello, " << n << '\n';
    }
}
```

Any questions before we move on?

Some common built-in types

- `char`: a single ASCII character, e.g. `char c = 'a'`
- `bool`: a boolean value, `true` or `false`
- `int`: a 32-bit* integer value
- `float`: a single-precision (32-bit*) floating point number
- `double`: a double-precision (64-bit*) floating point number

Integer types

- C++ has a large number of built-in integer types: `char`, `short int`, `int`, `long int`, `long long int`
- There are also `unsigned` varieties of all the above, which cannot hold negative values
- …or we can explicitly state we want a `signed` version!
- We can also omit the `int` part of the names of `short`, `long` and `long long`

Integer types

- Unfortunately, the size (in bytes) of the various integer types is not defined by the C++ standard, and varies between platforms

- In particular, `long` is 4 bytes on Windows, but 8 bytes on modern x86-64 Unix systems

- To get around this, the header `<cstdint>` defines aliases for fixed-size integer types, for example:

```
std::uint16_t u = 0; // 16-bit unsigned int
std::int64_t i = 0; // 64-bit signed int
uint32_t u2 = 0; // Can also omit std:: prefix
```

- Guideline: prefer using the fixed-size aliases, except for basic `int`s

std::string

- The C++ standard also defines a number of types (classes) which we can use to build our programs
- One which we'll be using frequently early in the course is `std::string`
- This is roughly like an array of `char`s, and defines operators so that it acts much like a built-in string type in other languages
- You need to include the header `<string>` to use `std::string`

std::vector

- Another type we'll be using frequently is `std::vector`

- This is like an *array* (collection) of some other type, to which we can add or remove values

- A `std::vector` is a *template*: this means we need to tell it what type we want it to contain, in angle brackets e.g.

```
std::vector<int> int_vec; // vector of ints
std::vector<std::string> str_vec; // vector of strings
```

- Templates are sort of like generics in other languages

- We'll learn more about `std::vector` later in the course

Raw arrays

- C++ has built-in arrays which it inherited from C
- These are also called “C arrays” or “raw arrays”
- These are declared as, for example

```
int arr[3] = {1, 2, 3}; // array of three ints
```

- WARNING: Raw arrays behave in very strange and unintuitive ways, and should be *avoided whenever possible*!
- Use `std::vector` instead (or `std::array`, which we’ll cover later)

Any questions before we move on?

Variables

- Dictionary definition: (roughly) “a named storage location for some data”
- In C++, every variable has a *type*, which dictates what sort of data it can hold
- The data currently held in a variable is called its *value*
- In C++, the *lifetime* of a variable is usually tied to the scope (block) in which it is declared

Declaring variables

- To declare a variable, we can say

```
type-name variable-name = initialiser;
```

- e.g.

```
int i = 0;
```

- Note: unlike other languages, the `new` keyword **should not** be used when declaring local variables

Declaring variables

- There are a couple of other initialisation forms which can also be used:
  - “Uniform initialisation style”, using curly braces:

    ```
    int i{0}; // i has value zero
    ```

  - “Function style”, using round brackets:

    ```
    int j(0); // j has value zero
    ```

- Which of these three forms is the right one to use depends on the type we are initialising, and often personal preference
- General guideline: prefer “=“ or curly-brace forms except in special cases

Declaring variables

- C++ in some cases allows you to declare a variable without an initial value, e.g.

```
int i; // legal, but bad!
```

- This is dangerous: if we try to read from an uninitialised variable, anything could happen!

- For safety, **always initialise your variables** with a sensible default value

Declaring variables

- When you declare a variable with an initialiser, C++ will check that the *type* of the initialiser is “compatible with” (*convertible to*) the type of the variable
- For example:

```
float f = 3; // Okay, can convert int -> float
std::string str = 3.0; // Error: cannot convert double -> std::string
```

- Beware: sometimes you can get unexpected (and undesired) conversions, for example:

```
int i = 3.9; // Legal, but i has value 3!
```

- Using the “curly brace” form guards against some of these unintended conversions

```
int i{3.9}; // Error, narrowing conversion
```

Variable declaration demo

- Open https://bit.ly/2jIMYYb

- Uncomment the line declaring `s2`. What errors do you get? Why?

- Experiment with changing between the three initialisation styles. Do you get any errors?

Any questions before we move on?

Constants

- We can declare a variable to be a *constant* using the keyword `const` in front of the type name, for example

```
const int i = 0;
```

- When declared like this, the value of `i` cannot be changed after it is declared

- This is helps reduce programming errors and (sometimes) allows better optimisation

- Guideline: make variables “`const` by default”, and mutable only when necessary

Constants

- It’s also possible to put the `const` keyword *after* the name of the type, for example `int const i = 0;`

- There are endless arguments over which is preferable. Guideline: choose one and be consistent!

- Const-ness is a property of a *type*: `int` and `const int` are *different types* in C++

- (Don’t worry too much about this distinction just yet.)

Type deduction

- C++11 added *type deduction* using the `auto` keyword, so we can say

```
auto variable-name = initialiser;
```

- e.g.

```
auto i = 0;
```

- Now the type of `i` is *deduced* from the type of its initialiser

- This also works with the other initialisation forms

Type deduction rules

- `auto` will never deduce a `const` type, for example

```
const int i = 3;
auto i2 = i; // i2 has type int, NOT const int
```

- (Also, `auto` will never deduce a reference type, as we'll see in a couple of weeks)

- We can say `const auto` when we want a variable of deduced type that we can never change

```
int j = 3;
const auto j2 = j; // j2 has type const int
```

Homework

- Complete the “drill” from Chaper 3 of the textbook (page 83)

Next time

- CLion install-fest!
- Things we didn’t have time for this week
- The C++ compilation model — header files, implementation files, libraries and linking

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…)