
# Module 2 — session 1

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"

# Bonus!

- Oli did a series of live-code demos about test-driven development (TDD)
- Find parts 1, 2, and 3 on our YouTube channel
- https://youtu.be/act1at7JeOU
- https://youtu.be/g9hyZHmmHRA
- https://youtu.be/ALpkqRbkBYM

# Last week

- Const references
- Namespaces
- End-of-module quiz

# This week

- Welcome to module 2!
- Defining our own structs in C++
- Defining member functions

# Revision: types

- In programming languages, a *type* is a way of giving meaning to some data

- The *type* of some data tells us what it represents and what we can do with it

- For example, we can multiply two numbers, but we cannot meaningfully multiply two strings

# Revision: types


- C++ has many built-in (“fundamental”) types, such as `int`, `float`, `double`, `bool` etc

- The standard library has lots more commonly-used types such as `std::string` and `std::vector`

- The language provides us with many tools to define our own types, which we’ll learn about ~~as the course progresses~~ today!

# Data structures


- A data structure is (abstractly) a way to organise the data used by your program.

- Designing data structures and the relationships between them is an essential element of programming

> “I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”
>
> — Linus Torvalds

# Data structures


- C++ provides us with several ways to build our own data structures
  - Structs/classes
  - Enumerations
  - Arrays
  - Unions

# Data structures


- C++ provides us with several ways to build our own data structures
  - Structs/classes
  - Enumerations
  - Arrays
  - Unions

# Structs


- A `struct` (or `class`) in C++ is a collection of *data members* (or *member variables*) together with *member functions* which operate on them

- We can define a new struct using the `struct` keyword

- The keywords `struct` and `class` mean almost exactly the same thing in C++. Today we'll talking about structs, but I'll often use the two terms interchangeably.

# Our first struct

- We can define a new struct using the `struct` keyword
- Inside the struct definition we list its *data members*, similarly to how we declare local variables in a function

```
struct point {
    int x = 0;
    int y = 0;
};
```

- A struct definition must always end with a semicolon!

# Our first struct

- The `struct` keyword always introduces a new type, distinct from any other type

- That means that two structs are different *types*, even if they have the same members

```
struct First {
    int i = 0;
};

struct Second {
    int i = 0;
};
```

- `First` and `Second` here are *different types*

# Our first struct


- C++ places no restrictions on the types of our member variables

- However, variables with reference type (e.g. `int`&) have surprising effects and are best avoided

- Consider using pointers (which we'll cover later) or `std::reference_wrapper` instead

# Our first struct


- We can create an *instance* of a `struct` in the same way as a built-in (*fundamental*) type like `int`

- We can access a struct's *members* using a `.` (dot) after the variable name, for example

```
point p{3, 7};
p.x = 8;
```

- We can use our struct anywhere that we can use a fundamental type, for example as a member variable of another struct, as a function parameter, or as the element type of a `std::vector`

# Exercise

- In the main.cpp file of a new CLion project, define a new struct called `Student`

- A `Student` should have two member variables, both of type `std::string`, named `first_name` and `surname`

- Write a function `void print_surname(const Student& s)` which prints the surname of the given student

- Extension: create a `std::vector` of students. Use a range-for loop to print the surname of each student

# Solution


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

struct Student {
    std::string first_name{};
    std::string surname{};
};

void print_surname(const Student& s)
{
    std::cout << s.surname << '\n';
}

int main()
{
    const Student tom{"Tom", "Breza"};
    print_surname(tom);

    std::vector<Student> students{
            {"Tom", "Breza"},
            {"Oli", "Ddin"},
            Student{“Tristan", "Brindle"}
    };

    for (const auto& s : students) {
        print_surname(s);
    }
}
```

# Member functions

- A *member function* is a function which belongs to a type, and (usually) operates on that type's *member variables*

- Non-member functions are often called *free functions*

- We can declare a *member function* using the same syntax as for non-member functions

```
struct point {
    bool equal_to(const point& other);

    int x = 0;
    int y = 0;
};
```

# Example


```
struct point {
    int x = 0;
    int y = 0;

    bool equal_to(const point& other) const {
        return x == other.x && y == other.y;
    }
};


point p{1, 2};
point q{4, 6};

if (p.equal_to(q)) {
    // Do something
}
```

# Member functions


- Within a member function, we can refer to *member variables* of the same struct *instance* without qualification

- If a member function is able to operate on a `const` *instance* of the class, we add the keyword `const` to the end of the member function declaration, for example:

```
struct point {
    bool equal_to(const point& other) const;

    int x = 0;
    int y = 0;
};
```

- Within a *const member function*, member variables behave as if they had been declared using the `const` keyword

# Homework

Your task for this week is to develop a simple record-keeping app for schools or universities. After each step, you should include tests to make sure everything works correctly.

1. Define a new struct `Student` with three member variables: a `first_name` and a `surname` (both strings) and an `id` (which should be an `int`)

2. Change the default initialiser for `Student::id` to use an incrementing counter. That is, the first `Student` instance you create should have id 1, the second id 2, and so on

3. Add a `print()` member function to `Student` which should print out the first name, the surname and the id number, separated by spaces

4. Define a new struct `ModuleRecord` with two member variables: a `Student` and an integer grade

5. Define a new struct `Module` which has two member variables: a `std::string` containing the module name, and a `std::vector<ModuleRecord>` of the grades for the module

6. Add an `add_record()` member function to your `Module` struct, which takes as arguments a `Student` and an integer grade. In the implementation of this member function, create a `ModuleRecord` and add it to the vector member.

7. Add a `print()` member function to your `Module` struct. For each element in the member vector, print out the student's first name, surname and id number followed by their grade

8. **Extension**: In the above `print()` function, print the records in descending order of their scores: that is, the highest-scoring student should have their name printed first, followed the second highest, and so on.

# 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…)