Support the face fuzz!

- http://mobro.co/olivercpp

Custom types — session 2

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

- Defining our own structs in C++
- Member functions

This week

- More on member functions: declarations and definitions
- Function overloading
- Operator overloading

Last week's 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.

Last week's homework

- https://github.com/CPPLondonUni/custom_types_week1_homework_soln

Revision: 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
- A `struct` definition always introduces a new type, even if it has the same members as another type

Revision: structs

- A *member variable* (sometimes called a *field* or a *data member*) is a variable that belongs to a struct

- We can declare member variables using the same syntax as for local variables in functions

```
struct MyStruct {
    int i = 0;
    float f = 3.14f;
};
```

- Tip: always use *member initialisers* for your member variables, unless you're sure you know what you're doing!

Revision: member functions

- A *member function* (sometimes called a *method*) is a function which belongs to a struct, and operates on that struct's *member variables*

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

- Member functions always operate on a particular *instance* of a struct

- 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;
};
```

Revision: member access

- Given an *instance* of a struct, we can access its members using the `.` (dot) operator, e.g

- When calling a member function, we can directly access the member variables of the instance we are operating on

- (If it helps, you can think of the “current instance” as being an extra, hidden function parameter passed to the member function)

- Within a member function, the keyword `this` refers to (a pointer to) the *current instance*

Revision: const member functions

- 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
- That means that within a const member function, the member variables are read-only

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
}
```

Any questions before we move on?

Structs and header files

- In order to use a struct, the compiler must have seen its *definition* — so that it knows what member variables and member functions it has

- For this reason, we usually place our struct definitions in *header files*, so they can be used by other parts of our code

Structs and header files

- A member function is *declared* as part of the struct…
- …but we may supply the *definition* of a member functions in a separate file
- Typically, write the definitions of simple member functions in the class definition itself ("inline"), while the definitions of more complex member functions are placed in a separate file

Example

```cpp
// dog.hpp
struct Dog {
    void bark();          // declaration
    void wag_tail() const; // declaration
    int get_num_legs() const { return 4; } // declaration and definition
    float hunger = 0.0f;
};

// dog.cpp
#include "dog.hpp"
#include <iostream>

void Dog::bark() {
    std::cout << "Woof!\n";
}

void Dog::wag_tail() const {
    std::cout << "*wags tail cutely*\n";
}
```

Exercise

- Create a new C++ executable project in CLion
- Add a new C++ class named `Point`. CLion will create the files `point.cpp` and `point.hpp` for you. Change the `class` keyword to `struct` in point.cpp.
- Add two member variables `x` and `y`, both of type int, to struct `Point`
- Add a *declaration* of a *const member function* named `equal_to`, taking as a parameter a const reference to another `Point`, and returning a `bool`
- Write the *definition* of your `equal_to` function in Point.cpp
- Add another member function not_equal_to. This time, *define* the member function inside the `Point` struct
- In `main.cpp`, write a test to check that your `equal_to()` and `not_equal_to()` member functions are working correctly

Solution

- Live demo (hopefully)

Function overloading

- In C++, we can have many functions with the same name, but different parameter lists
- For example:

```
void do_something(int i);
void do_something(double d);
```

- This is called function *overloading*

Function overloading

- When we call a function, the compiler tries to match the types of the *arguments* we supply with the *function parameters* declared in the function *signature*
- With overloaded functions, it will try to find the best matching function for the given arguments
- This process is called *overload resolution*

Example

```
void print(int i)
{
    std::cout << "One" << '\n';
}

void print(double f)
{
    std::cout << "Two" << '\n';
}


int i = 0;
print(i);
// Prints One

float f = 0.0;
print(f);
// Prints Two
```

•

Example

```cpp
void print(const int& i)
{
    std::cout << "One" << '\n';
}

void print(int& i)
{
    std::cout << "Two" << '\n';
}


int i = 0;
print(i);
// Prints Two

const int ci = 0;
print(ci);
// Prints One
```

•

Member function overloading

- Member functions may be overloaded, just like free functions

- For example:

```
struct Example {
    int do_something(int i); // (1)
    float do_something(float f); // (2)
};

Example e{};
e.do_something(1); // Calls (1)
e.do_something(3.14f); // Calls (2)
```

Member functions

- We can also *overload* member functions with different const-qualifiers to provide versions which do different things for const and non-const versions of the data type.

- For example:

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

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

point p{3, 4};
p.equal_to(point{3, 4}); // calls non-const equal_to()

const point cp{3, 4};
cp.equal_to(point{1, 2}); // calls equal_to() const
```

Any questions before
we move on?

Operator overloading

- C++ allows us to implement most *operators* for our custom types
- For example, we can define what the == operator means for our `Point` type
- This is called *operator overloading*
- We implement operator overloads by writing a function (member or non-member) named `operator@`, taking appropriate arguments

Operator overloading

- We write operator overloads using the syntax

```
bool operator==(const point& p, const point& q);
```

- Now we can compare two points using the usual == syntax, like built-in types

```
const Point p{3, 4};
const Point q{3, 4};
assert(p == q);
```

Operator overloading

- Almost all operators in C++ can be overloaded:

```
+     -     *     /     %     ^     &     |
~     !     ,     =
++    --    <<    >>    ==    !=    &&    ||
+=    -=    /=    %=    ^=    & =   |=    *=
<<=   >>=   [ ]   ( )   ->    ->*   new   delete
```

- Some operator overloads must be member functions, others may be written as free functions

- Operator overloading opens the door to doing many crazy things!

- Golden rule: only provide an operator overload when there is a “natural” meaning for that operator. “Do as the `int`s do”!

Operator overloading

- Question: which operators does it make sense to overload for our `point` class?

Homework

- For this week's homework we are going to write a little utility to convert measurements between various units.

  1. Create a new header file named `conversion.hpp`, and an accompanying `conversion.cpp`

  2. In `conversion.hpp`, create a struct named `Metres` and another named `Feet`. Both structs should have a single member of type `double`.

  3. Write a free function `to_feet()` which takes a single argument of type `Metres`, and returns a variable of type `Feet`, appropriately converted (1ft is 0.3048m).

  4. Write a corresponding `to_metres()` free function which performs the opposite conversion.

  5. Write a member function `add()` to struct `Metres`, taking an argument of type `Metres`. Update the stored distance by adding the new distance to it. What should the return type of this function be? Why? Write the definition in `conversion.cpp`. Write the equivalent member function in struct `Feet`.

  6. Write a free function named `to_string()` with two *overloads*: one for `Metres` and one for `Feet`.

  7. [Tricky] Write an overload of `Metres::add()` which takes an argument of type `Feet`. Write an overload of `Feet::add()` which takes an argument of type `Metres`.

  8. [Extension]: implement your add member functions as overloads of operator+=().

  9. [Further extension]: Implement *User Defined Literals* for metres and feet

Online resources

C++ London uni

- 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…)