Support the face fuzz!

- http://mobro.co/olivercpp

C++ London uni

Custom types — session 3

Tristan Brindle

Feedback

C++ London uni

- 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

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

This week

- More on operator overloading
- Constructors and destructors

Last week's homework

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

Last week's homework

- https://github.com/CPPLondonUni/custom_types_week2_homework_soln

Any questions before we move on?

Operators in C++

- C++ has many *operations* defined on built-in types
- For example, we compare two `bool`s for equality, or negate a (signed) `int`.
- We might write these as `a == b` or `-i`.
- Here, `==` and `-` are examples of *operators*
- Some operators take two arguments (*binary operators*), and some take a single argument (*unary operators*)
- Some operators have both *unary* and *binary* forms

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

- Most operator overloads may be written as either member functions or non-member functions

- A few may only be implemented as member functions

- When written as non-members, binary operator overloads have two parameters, and unary operators take one

- When written as members, binary operators have one parameter, and unary operators have zero

Operator overloading

	Unary Operator	Binary Operator
Non-member	One parameter	Two parameters
Member	Zero parameters	One parameter

Operator overloading

- One particularly useful application of operator overloading is to provide a stream operator, so we can print our type using `std::cout`.

- This must be written as a non-member function (why?)

```
std::ostream& operator<<(std::ostream& os, const Point& p)
{
    os << '(' << p.x << ", " << p.y << ')';
    return os;
}

std::cout << Point{1, 2} << '\n';
```

Operator overloading

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

Operator overloading

- Question: which operators does it make sense to overload for our `Point` class?
    - Equality comparison (==, !=)
    - Addition/subtraction of two points (+, -, +=, -=)
    - Streaming to `std::ostream`
    - Probably nothing else

Exercise

- Go to https://classroom.github.com/a/xHvzqHXa and clone the starter code
- The starter code contains the `point` struct from last week, and an implementation of `operator==`
- Tasks:
  - Implement operator!=
  - Implement (binary) operator+ and operator-
  - Implement operator+= and operator-=. What type should these functions return? (Think: do as the ints do!)
  - (Harder) Implement operator<< for output streams
  - Test all your operator overloads in your `main()` routine

Solution

https://github.com/CPPLondonUni/
week12_point_exercise/tree/ex1_solution

Any questions before we move on?

Resource management

- Often when writing programs we need to *acquire a resource* (for example memory) and *release* it later

```
void example() {
    auto res = acquire_resource();
    do_something_with(res);
    release(res);
}
```

- However, this is error-prone: as code gets more complex, we can easily forget to release a resource, or (attempt to) release it twice

- This is particularly problematic when using exceptions

Constructors and destructors

- The C++ language provides tools to help us:
  - *Constructors* are member functions which are automatically run when *constructing* an object
  - *Destructors* are member functions which are automatically run when *destroying* an object
- By acquiring resources in a constructor and releasing them in a destructor, we can use the C++ language rules to manage resources!

Resource management

- For example:

```
struct resource_handle {
    // ...
};

void example() {
    resource_handle res{};
    do_something_with(res);
}
```

- This pattern plays a central role in modern C++, and goes by the ~~silly~~ acronym *RAII*

- I prefer the term “scope based resource management”

- The C++ standard library provides some RAII handles for us, for example `std::vector` and `std::unique_ptr`

Constructors and destructors

- A *constructor* is a special kind of member function which is used when creating a new object

- The job of a constructor is to make the object ready for use

- A constructor is written as a member function with the **same name** as its enclosing class, and **no return type**

- For example:

```
struct Example {
    Example(int i); // ctor taking an int parameter
};
```

Constructors and destructors

- A destructor is a special member function which is used when *destroying* an object

- The job of a destructor is (typically) to release any resources acquired by the constructor

- We write a destructor as

```
struct Example {
    ~Example(); // Destructor for Example
};
```

Calling constructors

- We have seen that we can create a new instance of a struct by writing `T{arg1, arg2, …}`

- If the type has no *user-defined constructors* (and no non-public members or bases), then this will initialise every member in turn. This is called *aggregate initialisation*.

- Otherwise, this will (attempt to) call a matching constructor.

- For types which do have constructors, we can also use round brackets, i.e. `T(arg1, arg2, …)`

- Warning (1): Sometimes the `{}` and `()` forms do different things! (e.g. `std::vector`)

- Warning (2): Sometimes the round bracket form will be parsed as a function declaration(!)

Next week

- More on constructors:
  - Member initialiser lists
  - Explicit constructors
- Public and private member access

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…)