C++ London University Session 3

Tristan Brindle

Feedback

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- If you don’t know, please **ASK**

Lesson Plan

- Homework questions from last week
- Structs in C++
- Member functions
- Operator overloading
- Constructors
- Destructors
- Copy constructors
- Copy assignment operators
- The rule of zero

Any questions from last week's material?

“Homework” questions from last week

- What is wrong with the following function definition? How can we fix it?

```
std::string& get_hello()
{
    std::string str = "Hello";
    return str;
}
```

“Homework” questions from last week

- The function `get_hello()` returns a reference to a local `std::string` variable. When the function ends, the local variable is destroyed and the reference becomes invalid (“dangling”).

- We can fix this by returning the string *by value* instead

- Now the calling function will receive a copy of the local `str` variable (usually optimised by the compiler)

```
std::string get_hello()
{
    std::string str = "Hello";
    return str;
}
```

“Homework” questions from last week

- Write a function `fib(int n)` returning a vector of integers containing the first n Fibonacci numbers

“Homework” questions from last week

- My solution:

```
#include <vector>

std::vector<int> fib(int n)
{
    std::vector<int> output;
    output.push_back(0);
    output.push_back(1);

    for (int i = 2; i < n; i++) {
        const int last1 = output[i - 2];
        const int last2 = output[i - 1];
        output.push_back(last1 + last2);
    }

    return output;
}
```

“Homework” questions from last week

- Extension: modify `fib()` to allow the user to pass the initial “seed” values, defaulting to 0 and 1

“Homework” questions from last week

- My solution:

```
#include <vector>

std::vector<int> fib(int n, int seed1 = 0, int seed2 = 1)
{
    std::vector<int> output;
    output.push_back(seed1);
    output.push_back(seed2);

    for (int i = 2; i < n; i++) {
        const int last1 = output[i - 2];
        const int last2 = output[i - 1];
        output.push_back(last1 + last2);
    }

    return output;
}
```

“Homework” questions from last week

- Extension(2): In CLion, create a `libfibonacci` library containing your fib function, and a test program which ensures that the results are correct

“Homework” questions from last week

- See CMake project at

  https://github.com/CPPLondonUni/course_materials/tree/master/week2/homework

Any questions before we move on?

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 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;
};
```

Structs

- 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;
```

Member functions

- We can declare a member function using the same syntax as for non-member (“free”) functions

- Within a member function, we can refer to member variables without qualification

- If a member function can 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;
};
```

Member functions (2)

- We can *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
```

Member functions (3)

- We can write the definition of a member function inside the class (in which case it is also a declaration), or outside the class. For example:

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

    bool not_equal_to(const point& other) const
    {
        return !equal_to(other);
    }

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

bool point::equal_to(const point& other) const
{
    // implementation
}
```

Member functions (4)

- We normally need to put our class definitions in header files, so they can be used by other parts of our code

- It is common to write the implementations of simple member functions in the header file, and place the implementation of more complex member functions in a separate implementation file (.cpp file)

- To prevent linker errors, functions *defined* in a header file must use the keyword `inline`. Functions defined in-class are automatically `inline`.

Exercise

- 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
    - Write the definition of the `point` struct in `point.hpp`. Write the definition of the `point::equal_to(const point&) const` member function in point.cpp
    - In `main.cpp`, write a test to check that your `equal_to()` and `not_equal_to()` member functions are working correctly

Solution

- Live demo

Operator overloading

- C++ allows us to provide *operator overloads* for our custom types. This is done 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 (2)

- 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 `ints` do”!

Operator overloading (3)

- One particularly useful application of operator overloading is to provide a stream operator, so we can print our type using `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 (3)

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

Operator overloading (3)

- 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

- Implement the following operator overloads for our point class
  - == != + - += -=
  - << for output streams
- What should += and -= return? (Think: do as `ints` do!)
- Test your operator overloads in your `main()` routine

Solution

- Live demo

Constructors

- A member function with the same name as the class is called a *constructor*

- A constructor's job is to make the object ready for use, by performing any initialisation of member variables (and base classes) that might be required

- Constructors do not have a return type, and may not be declared `const`

```
struct example {
    example(int i, float f); // constructor
};
```

Constructors (2)

- Constructors may declared in a header and defined in an implementation file, just like normal member functions

- Constructors may be overloaded, just like normal member functions

- A constructor which takes no arguments is called a *default constructor*, and is one of the six *special member functions*

- If you do not write any constructors for your class, a default constructor will be provided for you by the compiler

Explicit constructors

- A constructor which takes a single argument can be used as an *implicit conversion* in some circumstances
- For example:

```
struct example {
    example(int i);
};

void func(const example& e);

func(3); // Not an error!
```

Explicit constructors (2)

- Implicit conversions like these can have surprising effects, and are usually not desired

- This can be prevented by using the keyword `explicit` in front of the constructor

- Get into the habit of declaring all single-parameter constructors `explicit` by default

```
struct example {
    explicit example(int i);
};

void func(const example& e);

func(3); // Now a compile error
func(example{3}); // Okay
```

Member initialisers

- A constructor's job is to make the class ready for use. This includes setting the initial values of member variables.

- To set the initial values, we can use a *member initialiser list*. This appears after the declaration, but before the body of the constructor

```
struct example {
    explicit example(int i, float f)
        : mem1{i}, mem2{f}
    {
        // assert(mem1 == i);
    }

    int mem1;
    float mem2;
};
```

Member initialisers (2)

- We can also provide initial values in our member variable declarations themselves, as we've seen in our `point` class

- The constructor initialiser list overrides the default values provided in the member declaration, if any

- **Ensure that all member variables are initialised**, either with a default member initialiser or in the constructor initialiser list.

Exercise

- Exercise:
  - In our example project, create a new class `line`, which contains two `points`, *start* and *end*
  - Add a constructor for class `line` taking two points, using them to initialise the *start* and *end* members
  - Add a default constructor which initialises *start* and *end* to {0, 0}
  - In your `main()` function, verify that your constructors are working correctly

Copy constructors

- A constructor which takes another object of the same type as an argument (as a const reference) is called a *copy constructor*

- A copy constructor is used by the compiler when a copy of a variable is required, for example when we pass variables to functions *by value*

```
void func(example e);

struct example {
    example(const example&); // copy constructor
};

example e1;
auto e2 = e1; // Uses copy constructor
func(e1); // Passes a copy of e1 to func() using copy constructor
```

Copy constructors (2)

- The copy constructor is another of the special member functions

- If you do not provide a copy constructor, the compiler will provide one for you, calling the copy constructor of each member variable in turn

- Most of the time, you do not need to write your own copy constructor

Exercise

- Exercise:
  - Add a copy constructor to class `line`
  - In your `main()` function, verify that the copy constructor is working correctly

Destructors

- A destructor is another *special member function* that runs when an object's lifetime ends

- A destructor's job is to clean up any resources that the object may be using

- You declare a destructor with a ~ in front of the class name. A destructor takes no arguments and has no return type. For example:

```
struct example {
    ~example();
};

{
    example e1{};
} // <- destructor is called here
```

Destructors (2)

- After the destructor body has run, the compiler will call the destructor of each member variable in turn.

- As with the other special members, the compiler will provide a destructor if you do not write one yourself.

- As with the other special member functions, you normally do not need to write your own destructor

Exercise

- Exercise:
  - Add a destructor to class `line`
  - Verify that the destructor is working correctly

Copy assignment operator

- The final special member that we'll be talking about today is the copy assignment operator

- This is an overload of `operator=` that takes a const reference to an object of the same type, returning a non-const reference to the assigned object

- The assignment operator is used when assigning to an *already constructed* variable. For example:

```
struct example {
    example& operator=(const example&);
};

example e1{};
example e2{};
e2 = e2; // calls assignment operator
example e3 = e2; // Copy constructor, not assignment
```

Copy assignment operator

- As with the other special members, the compiler will provide a copy assignment operator if you do not define an assignment operator yourself

- The compiler-provided version will simply assign each member variable in turn

- As with the other special members, you normally do not need to write your own copy assignment operator

Exercise

- You can probably guess :)

- (Add a copy assignment operator to class `line` and verify that it is working correctly.)

Solution to exercises

- Live demo

The Rule of Three

- The “rule of three” says that if you implement one of the special member functions (destructor, copy constructor, copy assignment operator), you almost certainly need to provide all three

The Rule of ~~Three~~ Five

- The “rule of three” says that if you implement one of the special member functions (destructor, copy constructor, copy assignment operator), you almost certainly need to provide all three

- C++11 added two new special member functions, a move constructor and move assignment operator (which we’ll talk about in a future session), turning this into the “rule of five”

The Rule of ~~Three~~ ~~Five~~ Zero

- The “rule of three” says that if you implement one of the special member functions (destructor, copy constructor, copy assignment operator), you almost certainly need to provide all three

- C++11 added two new special member functions, the move constructor and move assignment operator (which we’ll cover in a future session), turning this into the “rule of five”

- But much better is the rule of zero: in general, you should not provide any of the special member functions (other than perhaps a default constructor) yourself, but use the compiler-provided versions

Summary

- Structs in C++
- Member functions
- Operator overloading
- Constructors
- Destructors
- Copy constructors
- Copy assignment operators
- The rule of zero

Next week

- Member access (public, protected, private)
- Inheritance
- Virtual functions and polymorphism
- Pointers and smart pointers

Homework

- Write a new class `multiline`, which contains a vector of `points`. Which special members do you need to define for this class? Why?

- Add a member function `append(const point& p)` to your multiline class, which adds the given point to the end of the vector. Write member functions `front()` and `back()` which return the first and last points of the multiline respectively. Which of these member functions need to be declared `const`? Why?

- Which operator overloads (if any) should we add to our `multiline` class? Why? Implement those that you have chosen

- (Harder): write a `length()` member function for the `multiline` class which returns the total length of the line.

- Write a `test_multiline()` function which checks that your `multiline` class works as intended. Call this function from your `main()`.

- (Hint for testing `length()`: a multiline containing the points (0, 0), (3, 4), (15, 9) and (12, 5) in that order should have total length of 23)

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…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Github: github.com/CPPLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #cpplondon
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂