Support the face fuzz!

- http://mobro.co/olivercpp

Custom types — session 4

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

- More on operator overloading
- (A little about) constructors and destructors

This week

- Constructors and destructors
- Calling constructors
- Writing constructors
    - Member initialiser lists
    - Explicit constructors

Last week's homework

- 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

Last week's homework

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(!)

Writing constructors

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

- This includes acquiring any resources required by the object, and setting the initial values of any members

- Like ordinary member functions, we can write the constructor *declaration* and *definition* separately

- Like ordinary member functions, we can (and often do) have multiple *overloaded* constructors

Example

```
struct MyType {
    MyType() {}

    MyType(const std::string& s) {
        str = s;
    }

    MyType(int i);

    std::string str{};
};

MyType::MyType(int i) {
    str = std::to_string(i);
}

int main() {
    MyType m1{};
    MyType m2;
    MyType m3{"Hello World"};
    MyType m4(99);
    MyType m5();
}
```

Default constructors

- A constructor which can be called with no arguments is called a *default constructor*

- The default constructor is one of the *special member functions*

- If you don't write any constructors yourself, the compiler will (attempt to) provide a default constructor for you

- You can explicitly request a compiler-provided default constructor by writing `=default;` as the definition

Example

```
struct MyType {
    MyType() = default;

    MyType(const std::string& s) {
        str = s;
    }

    MyType(int i);

    std::string str{};
};

MyType::MyType(int i) {
    str = std::to_string(i);
}

int main() {
    MyType m1{};
    MyType m2;
    MyType m3{"Hello World"};
    MyType m4(99);
    MyType m5();
}
```

Exercise

- Go to `https://classroom.github.com/a/jtaOM5j_` and clone the starter code
- Follow the instructions in the README

Solution

- https://github.com/CPPLondonUni/week13_points_and_lines/tree/solution

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

- 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
```

Explicit constructors

- Like other functions, constructors can have *default arguments*

- This means that it's not always obvious when a constructor can take a single argument, and therefore be a candidate for implicit conversion

Member initialisers

- The job of a constructor is to make the an object ready for use. This includes setting the initial values of any member variables.

- One possible way of doing this is to set the value in the *body* of the constructor:

```
struct Example {
    Example() {
        i = 42;
    }

    int i;
};
```

Member initialisers

- However, C++ has a rule that says that initialisation of member variables (and base classes) is complete *before control enters the body of a constructor*
- (This prevents member variables from being in an "unconstructed" state)
- This means that setting the value of a member in a constructor body is *assignment*, not construction

Member initialisers

- All member variables (and base classes) are fully constructed before we reach the constructor body
- Usually this is via the member's *default constructor*
- By performing assignment in the constructor body, we are doing more work than we need to
- If a member is a type with *no* default constructor, we're in trouble!

Member initialisers

```
struct NoDefaultCtor {
    NoDefaultCtor(int, float);

    int get_int() const;
};

struct Example {
    NoDefaultCtor n;
    int i;

    Example();
};

Example::Example()
{
    n = NoDefaultCtor(1, 2);
    i = n.get_int();
}
```

Member initialisers

- We can instead initialise our member variables using a *member initialiser list* in our constructor
- This goes on the *definition* of the constructor, but *before the body*, and specifies how to initialise each member
- As always, by the time control enters the body of the constructor, all of our members are fully constructed

Member initialisers

```
struct NoDefaultCtor {
    NoDefaultCtor(int, float);

    int get_int() const;
};

struct Example {
    NoDefaultCtor n;
    int i;

    Example();
};

Example::Example()
    : n(1, 2.0f),
      i(n.get_int())
{
}
```

Member initialisers

- Important: member variables are **always** constructed in the order that they appear in your *struct definition*

- **ALWAYS**

- NOT in the order that you write them in the member init list

- For safety, **always** write the elements of the member init list in declaration order (most compilers today will warn you if you do not)

Default member initialisers

- We can also supply a *default member initialiser* when we define our class member
- You have already seen these in our Point class!
- If we do not mention a member variable in a constructor member init list, the default member initialiser will be used
- ALWAYS ensure that your members are correctly constructed!

Default member initialisers

```
struct NoDefaultCtor {
    NoDefaultCtor(int, float);

    int get_int() const;
};

struct Example {
    NoDefaultCtor n{1, 2.0};
    int i = n.get_int();

    Example(int i, float f);
    Example();
};

Example::Example(int i, float f)
    : n(i, f)
{}

Example::Example() = default;
```

Next week

- Public and private member access
- Enumerations
- End of module quiz

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…)