Custom types — session 5

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"

Last week

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

This week

- Explicit constructors
- Public and private member access
- End of module quiz

Last week's homework

`https://github.com/CPPLondonUni/week13_points_and_lines/`

The file `line.hpp` contains the definition of a `line` struct. A line is defined as two `point`s, the start and the end.

Exercise

- Write a constructor for `line` taking two `point`s as arguments. Use the given points to initialise the `start` and `end` members of `line`. You should write the declaration of the constructor in `line.hpp`, and the definition in `line.cpp`
- Add a default constructor to `line` which sets both members to the point (0, 0)
- In `main.cpp`, add assertions to `test_line()` to ensure your constructors are working correctly.
- Experiment with marking your constructors as `= default` or `= delete` (you will need to comment out the constructor definitions while you do this). What error messages, if any, did you see? Is this what you expected?
- (Optional, slightly harder): add a `length()` member function to `line` which returns the distance from the start point to the end point.

Last week's homework

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

Any questions before we move on?

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

Any questions before we move on?

Public and private member access

- Suppose we want to use a string that is guaranteed to contain only lower-case letters

- We can create a new type which has a `std::string` member, and which performs lower-case conversion in its *constructor*

- The type-safety rules of C++ will prevent us from mixing our lower-case-only strings with more general `std::strings`

Public and private member access


```cpp
#include <cctype>
#include <string>

struct lc_string {
    explicit lc_string(const std::string& s)
        : str(s)
    {
        for (char& c : str) {
            c = std::tolower(c);
        }
    }

    const std::string& get_string() const { return str; }

    void set_string(const std::string& s)
    {
        str = s;
        for (char& c : str) {
            c = std::tolower(c);
        }
    }

    std::string str;
};
```

Public and private member access

- Unfortunately, we have a problem
- There is nothing to stop a user from accessing our `str` member and setting it directly!

```
int main()
{
    lc_string hello{"HELLO"};
    std::cout << hello.get_string(); // prints "hello"

    hello.str = "HELLO";
    std::cout << hello.get_string(); // prints "HELLO"!
}
```

Public and private member access

- We say that our `lc_string` has the *invariant* that it is always lower-case
- Allowing unrestricted member access allows users to violate that invariant
- To avoid this, C++ allows us to mark struct members as `private`
- Private members may only be accessed by other members of the same struct

Public and private member access


```
struct lc_string {
    explicit lc_string(const std::string& s)
        : str(s)
    {
        for (char& c : str) {
            c = std::tolower(c);
        }
    }

    const std::string& get_string() const { return str; }

    void set_string(const std::string& s)
    {
        str_ = s;
        for (char& c : str) {
            c = std::tolower(c);
        }
    }

private:
    std::string str;
};

int main()
{
    lc_string hello{“HELLO"};
    hello.str = "HELLO"; // ERROR: attempt to access private member
}
```

Public and private member access

- We can use the `private` *access specifier* to designate struct members as *private* to our type

- Every member declared after `private:` will be inaccessible to users of our type, until the next access specifier (or the end of the struct declaration).

- There is a corresponding `public` access specifier as well, used for members which *are* accessible to end users

- (There is also a third kind of access level, `protected`, but we won't be talking about that today.)

- We can interleave `public:` and `private:` sections within our type definitions however we like

Public and private member access


```cpp
struct public_private_example {
    int i = 0;

private:
    float f = 3.14f;
    double d = 99.99;

public:
    std::string str = "Hello";
};

int main()
{
    public_private_example p{};

    p.i = 42; // okay, public
    p.str = "Goodbye"; // okay, public
    p.f = 2.718f; // ERROR, private
    p.d = 1.414; // ERROR, private
}
```

Private member functions

- As well as data members, we can declare *member functions* as being `private`

- This means that these member functions cannot be called by users of our type, but only by other member functions

- For example, in our `lc_string` we might wish to have a `make_lowercase()` member function which is an *implementation detail*

Private member functions


```cpp
struct lc_string {
    explicit lc_string(const std::string& s)
        : str(s)
    {
        make_lowercase();
    }

    const std::string& get_string() const { return str; }

    void set_string(const std::string& s)
    {
        str = s;
        make_lowercase();
    }

private:
    void make_lowercase()
    {
        for (char& c : str) {
            c = std::tolower(c);
        }
    }

    std::string str;
};


int main()
{
    lc_string hello{"HELLO"};
    hello.make_lowercase(); // ERROR: attempt to access private member
}
```

Structs and classes

- Unlike some other languages, the `struct` and `class` keywords in C++ mean almost exactly the same thing

- Both are used to declare a new *class type*

- The only difference is in their *defaults*

- Most notably, `struct`s default to `public` member access, and `class`es default to `private` member access

- We can override these defaults using access specifiers

- (If you're curious, the only other difference is that `struct`s default to *public inheritance*, whereas `class`es default to *private inheritance*.)

Structs and classes


```cpp
struct struct_example {
    int i = 0;

public:
    float f = 3.14f;

private:
    std::string str = "Hello";
};

class class_example {
    int i = 0;

public:
    float f = 3.14f;

private:
    std::string str = "Hello";
};

int main()
{
    struct_example s{};
    s.f = 2.71f; // Okay, public
    s.str = "Goodbye"; // ERROR: private
    s.i = 42; // Okay, public by default

    class_example c{};
    c.f = 2.71f; // Okay, public
    c.str = "Goodbye"; // ERROR: private
    c.i = 42; // ERROR: private by default
}
```

Friends

- Normally, private members of a class are only accessible by other members of that class

- However, it can occasionally be useful to allow the implementations of non-member functions to access private class members

- One common example is an output stream overload for debugging

- Within a class definition, the `friend` keyword can be used to grant particular free functions access to private members

Friends


```cpp
class friend_example {
    int i = 0;
    void private_fn();

    friend int friend_func(const friend_example& f);
};

int main()
{
    friend_example f{};
    friend_func(f);
}

int friend_func(const friend_example& f)
{
    f.private_fn(); // okay
    return f.i; //okay
}
```

Friends

- We can also use the `friend` keyword to grant friendship to another *class*. This is sometimes useful for logging, serialisation or debugging.

```
struct serialiser {};
class deserialiser {};

class friend_example {
    friend struct serialiser;
    friend class deserialiser;
};
```

- All of the other class's member function will be able to access our private member variables and functions

- Friendship is only one way! Granting friendship to a class will not allow you to access its non-public members.

- Friendship should be used rarely. Overuse of the `friend` keyword is usually a sign of a design issue — are you perhaps missing some public member functions?

Exercise

- Clone the starter code at https://classroom.github.com/a/575IlcBA

- Follow the instructions in the README file

Quiz time!

- https://tiny.cc/cppuni20181127

Next week

- A new module! Containers, iterators and the STL

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…)