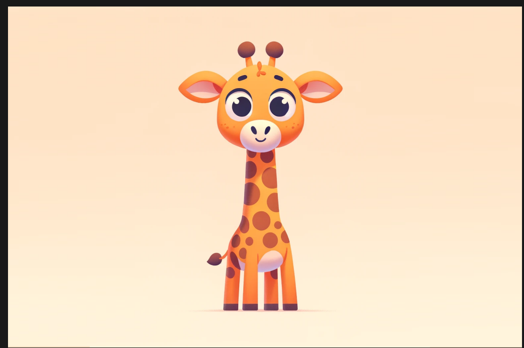
COMP6771

Class Types

Lecture 3.1

Author(s): Hayden Smith



(Download as PDF)

In This Lecture

- **Why?** 🤔
  - Defining our own types is a fundamental part of programming, so let's understand how things work
- **What?** 📰
  - Scope
  - Namespaces
  - Class Types

🔬 Scope

- The scope of a variable is the part of the program where it is accessible
  - Scope starts at variable definition
  - Scope (usually) ends at next "}"
  - You're probably familiar with this even if you've never seen the term
- Define variables as close to first usage as possible
- This is the opposite of what you were taught in first year undergrad
  - Defining all variables at the top is especially bad in C++

🔬 Scope

```cpp
#include <iostream>

int i = 1;
int main()
{
    std::cout << i << "\n";
    if (i > 0) {
        int i = 2;
        std::cout << i << "\n";
        {
            int i = 3;
            std::cout << i << "\n";
        }
        std::cout << i << "\n";
    }
    std::cout << i << "\n";
}
```

scope.cpp

🔬 Scope

Ways we create scopes:

- Classes
- Namespaces
- Functions
- Global
- Random braces

🤱 Object Lifetimes

- An object is a piece of memory of a specific type that holds some data
  - All variables are objects
  - Unlike many other languages, this does not add overhead
- Object lifetime starts when it comes in scope
  - "Constructs" the object
  - Each type has 1 or more constructor that says how to construct it
- Object lifetime ends when it goes out of scope
  - "Destructs" the object
  - Each type has a different "destructor" which tells the compiler how to destroy it

This is the behavior that primitive types follow, but you probably knew that intuitively. With classes, we tend to think a bit more explicitly about it.

🔨 Construction

Construction describes the process of allocating the memory and setting up for the creation of an object. Eg. https://en.cppreference.com/w/cpp/container/vector/vector

```cpp
#include <vector>

auto main() -> int
{
    auto v1 = std::vector<int>(1, 2);
    auto v2 = v1;
}
```

construction-basic.cpp

Construction can happen in many different ways

🔨 Construction

Construction can vary quite a bit!

Let's explore some different ways of constructing.

Let's also use this to understand scope better.

🔨 Construction

```cpp
#include <vector>

auto main() -> int
{
    // Always use auto on the left for this course, but you may see this elsewhere.
    std::vector<int> v11; // Calls 0-argument constructor. Creates empty vector.

    // There's no difference between these:
    // T variable = T{arg1, arg2, ...}
    // T variable{arg1, arg2, ...}
    auto v12 = std::vector<int> {}; // No different to first
    auto v13 = std::vector<int>(); // No different to the first

    {
        auto v2 = std::vector<int> { v11.begin(), v11.end() }; // A copy of v11.
        auto v3 = std::vector<int> { v2 }; // A copy of v2.
    } // v2 and v3 destructors are called here

    auto v41 = std::vector<int> { 5, 2 }; // Initialiser-list constructor {5, 2}
    auto v42 = std::vector<int>(5, 2); // Count + value constructor (5 * 2 => {2, 2, 2, 2, 2})
} // v11, v12, v13, v41, v42 destructors called here
```

construction-eg.cpp

🔨Construction Parantheses

- Generally use () to call functions, and to construct objects
  - () can only be used for functions, and can be used for either
  - There are some rare occasions these are different
    - Sometimes it is ambiguous between a constructor and an initialize list

🔨 Construction Parantheses

More examples

```
#include <iostream>

double f()
{
    return 1.1;
}

int main()
{
    // One of the reasons we do auto is to avoid ununitialized values.
    // int n; // Not initialized (memory contains previous value)

    int n21 {}; // Default constructor (memory contains 0)
    auto n22 = int {}; // Default constructor (memory contains 0)
    auto n3 { 5 };

    // Not obvious you know that f() is not an int, but the compiler lets it through.
    // int n43 = f();

    // Not obvious you know that f() is not an int, and the compiler won't let you (narrowing
    // conversion)
    // auto n41 = int{f()};

    // Good code. Clear you understand what you're doing.
    auto n42 = static_cast<int>(f());

    // std::cout << n << "\n";
    std::cout << n21 << "\n";
    std::cout << n22 << "\n";
    std::cout << n3 << "\n";
    std::cout << n42 << "\n";
}
```

construction-eg2.cpp

🍹 The Usefulness Of Object Lifetimes

Can you think of a thing where you always have to remember to do something when you're done?

- What happens if we omit f.close() here (assume similar behavior to c/java/python)?
- How easy to spot is the mistake
- How easy would it be for a compiler to spot this mistake for us?
  - How would it know where to put the f.close()?

```cpp
#include <algorithm>
#include <fstream>
#include <string>
#include <vector>

void ReadWords(const std::string& filename)
{
    std::ifstream f { filename };
    std::vector<std::string> words;
    std::copy(std::istream_iterator<std::string> { f }, {}, std::back_inserter(words));
    f.close();
}
```

lifetime.cpp

📛 Namespaces

Used to express that names belong together

```
// lexicon.hpp
namespace lexicon {
std::vector<std::string> read_lexicon(std::string const& path);

void write_lexicon(std::vector<std::string> const&, std::string const& path);
} // namespace lexicon
```

namespace-1.cpp

Prevent similar names from clashing

```
// word_ladder.hpp
namespace word_ladder {
std::unordered_set<std::string> read_lexicon(std::string const& path);
} // namespace word_ladder
```

namespace-2.cpp

📛 Namespaces

```cpp
// word_ladder.hpp
namespace word_ladder {
std::unordered_set<std::string> read_lexicon(std::string const& path);
} // namespace word_ladder
```

namespace-3.cpp

```cpp
// read_lexicon.cpp
namespace word_ladder {
std::unordered_set<std::string> read_lexicon(std::string const& path)
{
    // open file...
    // read file into std::unordered_set...
    // return std::unordered_set
}
} // namespace word_ladder
```

namespace-4.cpp

Namespaces

We can also nest them. We try and use top level instead of nesting where we can.

```cpp
namespace comp6771::word_ladder {
std::vector<std::vector<std::string>>
word_ladder(std::string const& from, std::string const& to);
} // namespace comp6771::word_ladder
```

namespace-nest1.cpp

```cpp
namespace comp6771 {
// ...

namespace word_ladder {
    std::vector<std::vector<std::string>>
    word_ladder(std::string const& from, std::string const& to);
} // namespace word_ladder
} // namespace comp6771
```

namespace-nest2.cpp

📛 Namespaces

You can create **unnamed** namespaces to provide a similar capability as C when it comes to functions local to a file.

We can also nest them

```cpp
namespace word_ladder {
namespace {
    bool valid_word(std::string const& word);
} // namespace
} // namespace word_ladder
```

namespace-unnamed.cpp

📛 Namespaces

We can also use **namespace aliases** to give a namespace a new name. It's often great for shortening nested namespaces.

```
namespace chrono = std::chrono;
```

Namespaces

It's important to **always fully-qualify** your namespaces even if you're already in that namespace.

```cpp
namespace word_ladder {
namespace {
    bool valid_word(std::string const& word);
} // namespace

std::vector<std::vector<std::string>>
generate(std::string const& from, std::string const& to)
{
    // ...
    auto const result = word_ladder::valid_word(word);
    // ...
}
} // namespace word_ladder
```

namespace-fully-qualify.cpp

⚽ Object-Oriented Programming (OOP)

Recap

- A class uses data abstraction and encapsulation to define an abstract data type:
  - **Abstraction**: separation of interface from implementation
    - Useful as class implementation can change over time
  - **Encapsulation**: enforcement of this via information hiding
- This abstraction leads to two key parts of the abstract data type:
  - **Interface**: the operations used by the user (an API)
  - **Implementation**: the data members the bodies of the functions in the interface and any other functions not intended for general use

⚽ Object-Oriented Programming (OOP)

Since you've completed COMP2511 (or equivalent), C++ classes should be pretty straightforward and at a high level follow very similar principles.

- A class:
  - Defines a new type
  - Is created using the keywords class or struct
  - May define some members (functions, data)
  - Contains zero or more public and private sections
  - Is instantiated through a constructor
- A member function:
  - must be declared inside the class
  - may be defined inside the class (it is then inline by default)
  - may be declared const, when it doesn’t modify the data members
- The data members should be private, representing the state of an object.

Member Access Control

This is how we support encapsulation and information hiding in C++

```cpp
class foo {
public:
    // Members accessible by everyone
    foo(); // The default constructor.

protected:
    // Members accessible by members, friends, and subclasses
    // Will discuss this when we do advanced OOP in future weeks.

private:
    // Accessible only by members and friends
    void private_member_function();
    int private_data_member_;

public:
    // May define multiple sections of the same name
};
```

member-access.cpp

⚙️ Constructor

Constructors behave very similar to other programming languages

```cpp
#include <iostream>

class myclass {
public:
    myclass(int i)
    {
        i_ = i;
    }
    int getval()
    {
        return i_;
    }

private:
    int i_;
};

int main()
{
    auto mc = myclass { 1 };
    std::cout << mc.getval() << "\n";
}
```

constructor-basic.cpp

📋 Constructor Initializer List

- The initialisation phase occurs before the body of the constructor is executed, regardless of whether the initialiser list is supplied
- A constructor will:

1. Construct all data members in order of member declaration (using the same rules as those used to initialise variables)
2. Construct any undefined member variables that weren't defined in step (1)
3. Execute the body of constructor: the code may assign values to the data members to override the initial values

📋 Constructor Initializer List

```cpp
#include <iostream>
#include <string>

class myclass {
public:
    myclass(int i)
        : i_ { i }
    {
    }
    int getval()
    {
        return i_;
    }

private:
    int i_;
};

int main()
{
    auto mc = myclass { 5 };
    std::cout << mc.getval() << "\n";
}
```

constructor-init.cpp

🐕 Constructor Logic Summary

- Constructors define how class data members are initalised
- A constructor has the same name as the class and no return type
- Default initalisation is handled through the default constructor
- Unless we define our own constructors the compile will declare a default constructor
  - This is known as the synthesized default constructor

```
for each data member in declaration order
if it has an used defined initialiser
  Initialise it using the used defined initialiser
else if it is of a built-in type (numeric, pointer, bool, char, etc.)
  do nothing (leave it as whatever was in memory before)
else
  Initialise it using its default constructor
```

🤙 Delegating Constructors

- A constructor may call another constructor inside the initialiser list
  - Since the other constructor must construct all the data members, do not specify anything else in the constructor initialiser list
  - The other constructor is called completely before this one.
  - This is one of the few good uses for default values in C++
    - Default values may be used instead of overloading and delegating constructors

🤙 Delegating Constructors

```cpp
#include <string>

class dummy {
public:
    explicit dummy(int const& i)
        : s_ { "Hello world" }
        , val_ { i }
    {
    }
    explicit dummy()
        : dummy(5)
    {
    }
    std::string const& get_s()
    {
        return s_;
    }
    int get_val()
    {
        return val_;
    }

private:
    std::string s_;
    const int val_;
};

auto main() -> int
{
    dummy d1(5);
    dummy d2 {};
}
```

constructor-delegate.cpp

🔫 Destructors

- Called when the object goes out of scope
- Why might destructors be handy?
  - Freeing pointers
  - Closing files
  - Unlocking mutexes (from multithreading)
  - Aborting database transactions
- noexcept states no exception will be thrown (we will cover this later)

```cpp
class MyClass {
    ~MyClass();
};
```

destructor-1.cpp

```cpp
MyClass::~MyClass()
{
    // Definition here
}
```

destructor-2.cpp

👆 This Pointer

- A member function has an extra implicit parameter, named this
  - This is a pointer to the object on behalf of which the function is called
  - A member function does not explicitly define it, but may explicitly use it
  - The compiler treats an unqualified reference to a class member as being made through the this pointer.
  - Generally we use a "_" suffix for class variables rather than a this-> to identify them

```cpp
#include <iostream>

class myclass {
public:
    myclass(int i)
    {
        i_ = i;
    }
    int getval()
    {
        return i_;
    }

private:
    int i_;
};

int main()
{
    auto mc = myclass { 1 };
    std::cout << mc.getval() << "\n";
}
```

this-1.cpp

```cpp
#include <iostream>

class myclass {
public:
    myclass(int i)
    {
        this->i_ = i;
    }
    int getval()
    {
        return this->i_;
    }

private:
    int i_;
};

int main()
{
    auto mc = myclass { 1 };
    std::cout << mc.getval() << "\n";
}
```

this-2.cpp

🔬 Class Scope

- Anything declared inside the class needs to be accessed through the scope of the class
  - Scopes are accessed using "::" in C++

```cpp
// foo.h

#include <istream>

class Foo {
public:
    // Equiv to typedef int Age
    using Age = int;

    Foo();
    Foo(std::istream& is);
    ~Foo();

    void member_function();
};
```

scope-1.h

```cpp
// foo.cpp
#include "scope-1.h"

Foo::Foo()
{
}

Foo::Foo(std::istream& is)
{
}

Foo::~Foo()
{
}

void Foo::member_function()
{
    Foo::Age age;
    // Also valid, since we are inside the Foo sc
    Age age;
}
```

scope-2.cpp

Class Scope

A simple example: C++ classes behaving how you'd expect

```cpp
#include <iostream>
#include <string>

class person {
public:
    person(std::string const& name, int const age);
    auto get_name() -> std::string const&;
    auto get_age() -> int const&;

private:
    std::string name_;
    int age_;
};

person::person(std::string const& name, int const age)
{
    name_ = name;
    age_ = age;
}

auto person::get_name() -> std::string const&
{
    return name_;
}

auto person::get_age() -> int const&
{
    return age_;
}

auto main() -> int
{
    auto p = person { "Hayden", 99 };
    std::cout << p.get_name() << "\n";
}
```

scope-3.cpp

Incomplete Types

- An incomplete type may only be used to define pointers and references, and in function declarations (but not definitions)
- Because of the restriction on incomplete types, a class cannot have data members of its own type.

```
struct node {
  int data;
  // Node is incomplete - this is invalid
  // This would also make no sense. What is sizeof(Node)
  node next;
};
```

- But the following is legal, since a class is considered declared once its class name has been seen:

```
struct node {
  int data;
  node* next;
};
```

🥷 Classes & Structs

- A class and a struct in C++ are almost exactly the same
- The **only** difference is that:
  - All members of a struct are public by default
  - All members of a class are private by default
  - **People have all sorts of funny ideas about this. This is the only difference**
- We use structs only when we want a simple type with little or no methods and direct access to the data members (as a matter of style)
  - This is a semantic difference, not a technical one
  - A std::pair or std::tuple may be what you want, though

```cpp
class foo { int member_ };
```

```cpp
struct foo { int member_ };
```

👀 Explicit Keyword

- If a constructor for a class has 1 parameter, the compiler will create an implicit type conversion from the parameter to the class
- This may be the behaviour you want (but usually not) You have to opt-out of this implicit type conversion with the explicit keyword

```cpp
#include <iostream>

class age {
public:
    age(int age)
        : age_ { age }
    {
    }

    auto getAge() { return age_; }

private:
    int age_;
};

auto main() -> int
{
    // Explicitly calling the constructor
    age a1 { 20 };

    // Explicitly calling the constructor
    age a2 = age { 20 };

    // Attempts to use an integer
    // where an age is expected.
    // Implicit conversion done.
    // This seems reasonable.
    age a3 = 20;

    (void)a1;
    (void)a2;
    std::cout << a3.getAge() << "\n";
}
```

explicit-1.cpp

```cpp
#include <vector>

class intvec {
public:
    // This one allows the implicit conversion
    // intvec(std::vector<int>::size_type length)
    // : vec_(length, 0);

    // This one disallows it.
    explicit intvec(std::vector<int>::size_type length)
        : vec_(length, 0)
    {
    }

private:
    std::vector<int> vec_;
};

auto main() -> int
{
    int const size = 20;
    // Explictly calling the constructor.
    intvec container1 { size }; // Construction
    intvec container2 = intvec { size }; // Assignment

    // Implicit conversion.
    // Probably not what we want.
    // intvec container3 = size;
}
```

explicit-2.cpp

💾 Const Members

- Member functions are by default only callable by non-const objects
  - You can declare a const member function which is valid on const objects and non-const objects
  - A const member function may only modify mutable members
    - A mutable member should mean that the state of the member can change without the state of the object changing
    - Good uses of mutable members are rare
    - Mutable is not something you should set lightly
    - One example where it might be useful is a cache

💾 Const Members

```cpp
#include <iostream>
#include <string>

class person {
public:
    person(std::string const& name)
        : name_ { name }
    {
    }
    auto set_name(std::string const& name) -> void
    {
        name_ = name;
    }
    auto get_name() -> std::string const&
    {
        return name_;
    }

private:
    std::string name_;
};

auto main() -> int
{
    person p1 { "Hayden" };
    p1.set_name("Chris");
    std::cout << p1.get_name() << "\n";

    person const p2 { "Hayden" };
    // p2.set_name("Chris"); // WILL NOT WORK... WHY NOT?
    // std::cout << p2.get_name() << "\n"; // WILL NOT WORK... WHY NOT?
}
```

const-members.cpp

🔌 Static Members

- Static members belong to the class (i.e. every object), as opposed to a particular object.
- These are essentially globals defined inside the scope of the class
  - Use static members when something is associated with a class, but not a particular instance
  - Static data has global lifetime (program start to program end)

🔌 Static Members

```cpp
#include <string>

class user {
public:
    user(std::string const& name)
        : name_ { name }
    {
    }
    auto valid_name(std::string const& name) -> bool
    {
        return name.length() < 20;
    }

private:
    std::string name_;
};

auto main() -> int
{
    auto n = std::string { "Santa Clause" };
    auto u = user { n };
    if (u.valid_name(n)) {
        user user1 { n };
    }
}
```

static-1.cpp

```cpp
#include <string>

class user {
public:
    user(std::string const& name)
        : name_ { name }
    {
    }
    static auto valid_name(std::string const& name) -> bool
    {
        return name.length() < 20;
    }

private:
    std::string name_;
};

auto main() -> int
{
    auto n = std::string { "Santa Clause" };
    if (user::valid_name(n)) {
        user user1 { n };
    }
}
```

static-2.cpp

Static member fields are usually defined outside of the class scope. This will be explored in your tutorial

🛀 Defaults

The synthesized default constructor

- Is generated for a class only if it declares no constructors
- For each member, calls the in-class initialiser if present
  - Otherwise calls the default constructor (except for trivial types like int)
- Cannot be generated when any data members are missing both in-class initialisers and default constructors

```cpp
class A {
  int a_;
};
```

```cpp
class B {
  B(int b): b_{b} {

  }
  int b_;
};
```

```cpp
int main() {
  int i_{0}; // in-class initialiser
  int j_; // Untouched memory
  A a_;
  // This stops default constructor
  // from being synthesized.
  B b_;
};
```

🛀 Defaults

Deleting unused default member fns

- Revise "The synthesized default constructor"
- There are several special functions that we must consider when designing classes
- Ask yourself the question:
  - Does it make sense to have this default member function?
    - **Yes**: Does the compile synthesised function make sense?
      - **No**: write your own definition
      - **Yes**: write `<function declaration> = default;`
    - **No**: write `<function declaration> = delete;`

🛀 Defaults

Default & Delete

Let's look at an example regarding the copy constructor

```cpp
#include <vector>

class intvec {
public:
    // This one allows the implicit conversion
    explicit intvec(std::vector<int>::size_type length)
        : vec_(length, 0)
    {
    }
    // intvec(intvec const& v) = default;
    // intvec(intvec const& v) = delete;

private:
    std::vector<int> vec_;
};

auto main() -> int
{
    intvec a { 4 };
    // intvec b{a}; // Will this work?
}
```

default-delete.cpp

👂 Feedback

Or go to the form here.