COMP6771
Advanced C++ Programming

Week 3.1
Class Types

Today we are covering

- Scope
- Class Types
- Assignment 1 / Other:
  - Creating a new directory (+CMakeLists)
  - Creating and running a new single file
  - Committing those changes with git
  - Questions about ass1 (might do at end)

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++

```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";
}
```

lecture-3/demo301-scope.cpp

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

- Eg. https://en.cppreference.com/w/cpp/container/vector/vector
- 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

```cpp
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
```

lecture-3/demo302-construction.cpp

Construction

- Also works for your basic types
  - But the default constructor has to be manually called
    - This potential bug can be hard to detect due to how function stacks work (variable may happen to be 0)
    - Can be especially problematic with pointers

```cpp
#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";
}
```

lecture-3/demo303-construction2.cpp

Why are object lifetimes useful?

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()?

```
1 void ReadWords(const std::string& filename) {
2   std::ifstream f{filename};
3   std::vector<std::string> words;
4   std::copy(std::istream_iterator<std::string>{f}, {}, std::back_inserter{words});
5   f.close();
6 }
```

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
```

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
```

Prevent similar names from clashing.

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

Namespaces

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

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

Nested namespaces

```
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 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
```

Prefer top-level and occasionally two-tier namespaces to multi-tier.

It's okay to own multiple namespaces per project, if they logically separate things.

Unnamed namespaces

In C you had static functions that made functions local to a file.

C++ uses "unnamed" namespaces to achieve the same effect.

Functions that you don't want in your public interface should be put into unnamed namespaces.

Unlike named namespaces, it's okay to nest unnamed namespaces.

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

Namespace aliases

Gives a namespace a new name. Often good for shortening nested namespaces.

```
namespace chrono = std::chrono;

namespace views = ranges::views;
```

Always fully qualify your function calls...

There are certain complex rules about how overload resolution works that will surprise you, so it's a best practice to **always fully-qualify** your function calls.

```
int main() {
    auto const x = 10.0;
    auto const x2 = std::pow(x, 2);

    auto const ladders = word_ladder::generate("at", "it");

    auto const x2_as_int = gsl_lite::narrow_cast<int>(x2);
}
```

Using a namespace alias counts as "fully-qualified" only if the alias was fully qualified.

...even if you're in the same namespace

There are certain complex rules about how overload resolution works that will surprise you, so it's a best practice to **always fully-qualify** your function calls.

```
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
```

Using a namespace alias counts as "fully-qualified" only if the alias was fully qualified.

...even if you're in the same nested namespace

There are certain complex rules about how overload resolution works that will surprise you, so it's a best practice to **always fully-qualify** your function calls.

```
namespace word_ladder::something::very_long {
    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::something::very_long::valid_word(word);
        // ...
    }
} // namespace word_ladder
```

Using a namespace alias counts as "fully-qualified" only if the alias was fully qualified.

What is OOP

- A class uses data abstraction and encapsulation to define an 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
  - **Abstraction**: separation of interface from implementation
    - Useful as class implementation can change over time
  - **Encapsulation**: enforcement of this via information hiding

Example: Bookstore :

- bookstore.h (interface)
- bookstore.cpp (implementation)
- bookstore_main.cpp (knows the interface).

C++ classes

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++

```
1  class foo {
2   public:
3    // Members accessible by everyone
4    foo(); // The default constructor.
5
6   protected:
7    // Members accessible by members, friends, and subclasses
8    // Will discuss this when we do advanced OOP in future weeks.
9
10  private:
11   // Accessible only by members and friends
12   void private_member_function();
13   int private_data_member_;
14
15  public:
16   // May define multiple sections of the same name
17 };
```

A simple example

C++ classes behave how you expect

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

class person {
public:
        person(std::string const& name, int 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 {
        person p("Hayden", 99);
        std::cout << p.get_name() << "\n";
}
```

lecture-3/demo304-classbasic.cpp

Classes and structs in C++

- 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

```
1 class foo {
2  int member_; // default private
3 };
```

```
1 struct foo {
2  int member_; // default public
3 };
```

Class Scope

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

```h
// foo.h

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

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

  void member_function();
};
```

```cpp
// foo.cpp
#include "foo.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 scope.
  Age age;
}
```

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;
};
```

Constructors

- 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 in-class initialiser
    Initialise it using the in-class 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
```

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

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

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

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

Constructor initialiser list

```cpp
#include <string>

class nodefault {
public:
        explicit nodefault(int i)
        : i_{i} {}

private:
        int i_;
};

int const b_default = 5;

class b {
        // Constructs s_ with value "Hello world"
        explicit b(int& i)
        : s_{"Hello world"} , const_{b_default} , no_default_{i} , ref_{i} {}

        // Doesn't work - constructed in order of member declaration.
        /*explicit b(int& i)
        : s_{"Hello world"} , const_{5} , ref_{i} , no_default_{ref_} {}*/

        /*explicit b(int& i) {
           // Constructs s_ with an empty string, then reassigns it to "Hello world"
           // Extra work done (but may be optimised out).
           s_ = "Hello world";

           // Fails to compile (modifying a const object).
           const_string_ = "Goodbye world";
           // Fails to compile (references *must* be initialized in the constructor).
           ref_ = i;
           // This is fine, but it can't construct it initially.
           no_default_ = nodefault{1};
        }*/

        std::string s_;
        // All of these will break compilation if you attempt to put them in the body.
        const int const_;
        nodefault no_default_;
        int& ref_;
};
```

- 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. Execute the body of constructor: the code may **assign** values to the data members to override the initial values

lecture-3/demo306-initlist.cpp

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{};
}
```

lecture-3/demo307-deleg.cpp

Destructors

- Called when the object goes out of scope
  - What might this be handy for?
- Why might destructors be handy?
  - Freeing pointers
  - Closing files
  - Unlocking mutexes (from multithreading)
  - Aborting database transactions

```
1 class MyClass {
2   ~MyClass() noexcept;
3 };
```

```
1 MyClass::~MyClass() noexcept {
2   // Definition here
3 }
```

Explicit type conversions

- 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

```
1  class age {
2  public:
3          age(int age)
4          : age_{age} {}
5
6  private:
7          int age_;
8  };
9
10 auto main() -> int {
11         // Explicitly calling the constructor
12         age a1{20};
13
14         // Explicitly calling the constructor
15         age a2 = age{20};
16
17         // Attempts to use an integer
18         // where an age is expected.
19         // Implicit conversion done.
20         // This seems reasonable.
21         age a3 = 20;
22 }
```

lecture-3/demo310-explicit1.cpp

```
1  #include <vector>
2
3  class intvec {
4  public:
5          // This one allows the implicit conversion
6          // intvec(std::vector<int>::size_type length)
7          // : vec_(length, 0);
8
9          // This one disallows it.
10         explicit intvec(std::vector<int>::size_type length)
11         : vec_(length, 0) {}
12
13 private:
14         std::vector<int> vec_;
15 };
16
17 auto main() -> int {
18         int const size = 20;
19         // Explictly calling the constructor.
20         intvec container1{size}; // Construction
21         intvec container2 = intvec{size}; // Assignment
22
23         // Implicit conversion.
24         // Probably not what we want.
25         // intvec container3 = size;
26 }
```

lecture-3/demo310-explicit2.cpp

Const objects

- Member functions are by default only be possible on non-const objects
  - You can declare a const member function which is valid on 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 member functions

```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 p1{"Hayden"};
        p1.set_name("Chris"); // WILL NOT WORK... WHY NOT?
        std::cout << p1.get_name() << "\n"; // WILL NOT WORK... WHY NOT?
}
```

lecture-3/demo308-const.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.
- For the next few slides, we'll be taking a look at the BookSale example in the course repo

```
1 class foo {
2  public:
3         foo(int const miles) {
4         this->kilometres_ = miles / 1.159;
5     }
6  private:
7         int kilometres_; // default private
8 };
```

```
1 class foo {
2  public:
3         foo(int const miles) {
4         kilometres_ = miles / 1.159;
5     }
6  private:
7         int kilometres_; // default private
8 };
```

Static members

- Static functions and 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)

```
1  // For use with a database
2  class user {
3    public:
4          user(std::string const& name) : name_{name} {}
5          static auto valid_name(std::string const& name) -> bool {
6                  return name.length() < 20;
7          }
8    private:
9          std::string name_;
10 }
11
12 auto main() -> int {
13         auto n = std::string{"Santa Clause"};
14         if (user::valid_name(n)) {
15                 user user1{n};
16         }
17 }
```

lecture-3/demo309-static.cpp

OOP design

- There are several special functions that we must consider when designing classes
- For each of these functions, ask yourself:
  - Is it sane to be able to do this? Does it have a defined, obvious implementation?
- If the answer to either of these is no, write "<function declaration> = delete;"
- Then ask yourself "is this the behaviour of the compiler-synthesized one"
  - If so, write "<function declaration> = default;"
  - If not, write your own definition
- Revise "The synthesized default 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?
}
```

lecture-3/demo311-delete.cpp