COMP6771 Week 3

Object-Oriented Programming

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++

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

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 you cannot do this (see last example)
    - Sometimes it is ambiguous between a constructor and an initialize list

```cpp
int main() {
  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, ...}
  std::vector<int> v12{}; // No different to first
  std::vector<int> v13 = std::vector<int>(); // No different to the first
  std::vector<int> v14 = std::vector<int>{}; // No different to the first

  std::vector<int> v3{v2.begin(), v2.end()}; // constructed with an iterator
  std::vector<int> v4{v3}; // Constructed off another vector

  std::vector<int> v51{5, 2}; // Initialiser-list constructor {5, 2}
  std::vector<int> v52(5, 2); // Count + value constructor (5 * 2 => {2, 2, 2, 2, 2})
}
```

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
int main() {
  int n; // not constructed (memory contains previous value)
  int n2{}; // Default constructor (memory contains 0)
  int n3{5};

  // This version is nice because it gives us an error.
  int n4{5.5};
  // You need to explictly tell it you want this.
  int n6{static_cast<int>(5.5)};

  // Not so nice. No error
  int n5 = 5.5;
}
```

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()?

```cpp
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();
}
```

Noexcept

- Exceptions will be covered in week 5, but the short version is that they are recoverable, but critical errors
- A noexcept-specified function tells the compiler not to generate recovery code
- An exception thrown in a noexcept function will terminate your program
- Use noexcept to guarantee that callers needn't worry about exception-handling.
- You can use noexcept to say that you don't mind your whole program ending if something goes wrong in this function.

Destructors

- Call when the object goes out of scope
  - What might this be handy for?
  - Does not occur for reference objects (why?)
- Marked noexcept (why?)
- Why might destructors be handy?

Destructors

- Called when the object goes out of scope
  - What might this be handy for?
  - Does not occur for reference objects (why?)
- Marked noexcept (why?)
- Why might destructors be handy?

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

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

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
  - Encapsulation: enforcement of this via information hiding
  - Example: Bookstore - bookstore.h (interface), bookstore.cpp (implementation), user code (knows the interface).

C++ classes

- 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.

Abstraction and encapsulation

- Abstraction is separating the interface from the implementation
- Encapsulation is hiding details about class representation and implementation
  - An object’s state can only be accessed/modified via the public interface

Advantages:

- Object state is protected from user-level errors
  - Users can't break invariants by changing something
- Class implementation may evolve over time
  - If you change a variable or a private function, users don't need to change anything

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;
};
```

Member access control

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

```cpp
class Foo {
 public:
  // Members accessible by everyone
  Foo();

 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 PrivateMemberFunction();

  int private_data_member_;

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

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

Friends

- A class may declare friend functions or classes
  - Those functions / classes are non-member functions that may access private parts of the class
  - This is, in general, a bad idea, but there are a few cases where it may be required
    - Nonmember operator overloads (will be discussing soon)
    - Related classes
      - A Window class might have WindowManager as a friend
      - A TreeNode class might have a Tree as a friend
      - Container could have iterator_t<Container> as a friend
        - Though a nested class may be more appropriate
  - Use friends when:
    - The data should not be available to everyone
    - There is a piece of code very related to this particular class

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

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

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

  void MemberFunction();
};
```

```cpp
// foo.cpp
#include "foo.h"

Foo::Foo() {
}

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

Foo::~Foo() {
}

void Foo::MemberFunction() {
  Foo::Age age;
}
```

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.
  - The **this** pointer always has top-level const
- For the next few slides, we'll be taking a look at the BookSale example in the course repo

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.
  - The **this** pointer always has top-level const
- For the next few slides, we'll be taking a look at the BookSale example in the course repo

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
- Let's make the BookSale class const correct

Are the following correct

Are the following correct?

```
Sales_data a{"Harry Potter"};
Sales_data b{"Harry Potter"};

a.combine(b).print(std::cout);
a.print(std::cout).combine(b);
```

Are the following correct

Are the following correct?

```
Sales_data a{"Harry Potter"};
Sales_data b{"Harry Potter"};

a.combine(b).print(std::cout);
a.print(std::cout).combine(b);
```

- The combine/print is fine
- The print/combine fails since print returns a const reference through which we cannot call a nonconst member
- Four possible ways to get it to compile. Discuss.
    - Make combine a const function
    - Make print a non-const function
    - Add an overload to print for non-const
    - Change the user code

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
- Is incorrect if some members of built-in types have no in-class initialisers
- Cannot be generated when any data members are missing default constructors

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

```cpp
class C {
  int i{0}; // in-class initialiser
  int j; // Untouched memory
  A a;
  // This stops default constructor
  // from being synthesized.
  B b;
};
```

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

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

Constructor initialiser list

```cpp
class NoDefault {
  NoDefault(int i);
}

class B {
  // Constructs s_ with value "Hello world"
  B(int& i): s_{"Hello world"}, const_{5}, no_default{i}, ref_{i} {}
  // Doesn't work - constructed in order of member declaration.
  B(int& i): s_{"Hello world"}, const_{5}, ref_{i}, no_default{ref_} {}
  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
    const_string_ = "Goodbye world";
    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_;
};
```

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

Static members

- Both data and function members may be declared static
- 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)

```cpp
// For use with a database
class User {
  static std::string table_name;
  static std::optional<User> query(const std::string& username);

  void commit();
  std::string username;
}

User user = *User::query("Alice");
user.username = "Bob"
User::commit(); // fails to compile (commit is not static)
user.commit();

std::cout << User::table_name;
std::cout << User::username; // Fails to compile
```

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

```cpp
class Age {
  Age(int age);
};

// Explicitly calling the constructor
Age age{20};
// Attempts to use an integer
// where an age is expected.
// Implicit conversion done.
// This seems reasonable.
Age age = 20;
```

```cpp
class IntVec {
  // This one allows the implicit conversion
  IntVec(int length): vec_(length, 0);

  This one disallows it.
  explicit IntVec(int length): vec_(length, 0);

  std::vector<int> vec_;
};

// Explictly calling the constructor.
IntVec container{20};

// Implicit conversion.
// Probably not what we want.
IntVec container = 20;
```

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 well-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
- Let's discuss these questions for these types over the next few slides:
  - std::vector
  - Mutex
  - Pointer

Copying constructor

- Constructs one object to be a copy of another
- The compiler-generated copy-constructor just calls each member's copy constructor in order of declaration

```cpp
class T {
  T(const T&);
};
```

Copying assignment

- Like a copy constructor, but the destination is already constructed
- Requires destroying the old data, and constructing the new data
- Copy-and-swap idiom is an elegant way of doing this
  - It constructs then destructs. Since construction might fail, it should go first
  - Requires move assignment to be defined
- Takes in an lvalue
- Compiler-generated one performs memberwise copy-assignment operator

```
class T {
  T& operator=(const T& original);

  // The copy-and-swap idiom
  T& operator=(T copy) {
    std::swap(*this, copy);
    return *this;
  }
};
```

```
MyClass base;
MyClass copy_constructed = base;

MyClass copy_assigned;
copy_assigned = base;
```

Rvalue references

- Rvalue references look like T&& (lvalue is T&)
- An lvalue denotes an object whose resource cannot be reused
  - Most objects (eg. variable, variable[0])
  - Once the lvalue reference goes out of scope, it may still be needed
- An rvalue denotes an object whose resources can be reused
  - eg. Temporaries (MyClass object in f(MyClass{}))
  - When someone passes it to you, they don't care about it once you're done with it

```
void f(MyClass&& x);
```

- “The object that x binds to is YOURS. Do whatever you like with it, no one will care anyway”
- Like giving a copy to f... but without making a copy.

Rvalue references

```
void inner(int&& value) {
  ++value;
  std::cout << value << '\n';
}

void outer(int&& value) {
  inner(value); // This fails? Why?
  std::cout << value << '\n';
}

int main() {
  f1(1); // This works fine.
  int i;
  f2(i); // This fails because i is an lvalue.
}
```

- An rvalue reference formal parameter means that the value was disposable from the caller of the function
  - If outer modified value, who would notice / care?
    - The caller (main) has promised that it won't be used anymore
  - If inner modified value, who would notice / care?
    - The caller (outer) has never made such a promise.
    - An rvalue reference parameter is an lvalue inside the function

std::move

```
// Looks something like this.
T&& move(T& value) {
  return static_cast<T&&>(value);
}
```

- Simply converts it to an rvalue
  - This says "I don't care about this anymore"
  - All this does is allow the compiler to use rvalue reference overloads

```
void inner(int&& value) {
  ++value;
  std::cout << value << '\n';
}

void outer(int&& value) {
  inner(std::move(value));
  // Value is now in a valid but unspecified state.
  // Although this isn't a compiler error, this is bad code.
  // Don't access variables that were moved from, except to reconstruct them.
  std::cout << value << '\n';
}

int main() {
  f1(1); // This works fine.
  int i;
  f2(std::move(i));
}
```

Move constructor

- Always should be declared noexcept
- Unless otherwise specified, objects that have been moved from are in a valid but unspecified state
- Will likely be faster than the copy constructor
- Compiler-generated one performs memberwise move-construction

```
class T {
  T(T&&) noexcept;
};
```

Move assignment

- Always should be declared noexcept
- Like the move constructor, but the destination is already constructed
- Compiler-generated one performs memberwise move-assignment

```
class T {
  T& operator=(T&&) noexcept;
};
```

Object lifetimes

To create safe object lifetimes in C++, we always attach the lifetime of one object to that of something else

- A variable in a function is tied to its scope
- A data member is tied to the lifetime of the class instance
- An element in a std::vector is tied to the lifetime of the vector
- A heap object should be tied to the lifetime of whatever object created it
- Examples of bad programming practice
  - An owning raw pointer is tied to nothing
  - A C-style array is tied to nothing
- **Strongly recommend** watching the first 44 minutes of Herb Sutter's cppcon talk "Leak freedom in C++... By Default"

Constructing wrapper types

```cpp
class MyClass {
  MyClass();
  MyClass(int);
  MyClass(const MyClass&);
  MyClass(MyClass&&);
  int GetValue();
};

// calls default constructor
std::optional<MyClass> opt1 = std::make_optional<MyClass>();
// calls int constructor
std::optional<MyClass> opt2 = std::make_optional<MyClass>(5);
// calls copy constructor
std::optional<MyClass> opt3 = *opt1;
// calls move constructor
std::optional<MyClass> opt4 = std::move(*opt1);
opt4->GetValue();

// Similar for make_unique and make_shared, but have to manually move / copy values
std::shared_ptr<MyClass> sp1 = std::make_shared<MyClass>();
std::shared_ptr<MyClass> sp2 = sp1;
std::shared_ptr<MyClass> sp3 = std::move(sp1);
MyClass* p2 = sp1.get();

std::unique_ptr<MyClass> up1 = std::make_unique<MyClass>(*sp1);
std::unique_ptr<MyClass> up2 = *up1; // But have to manually move / copy values like above
std::unique_ptr<MyClass> up3 = up1; // Fails (no copy constructor)
std::unique_ptr<MyClass> up4 = std::move(up1);
MyClass* p1 = up1.get();
up1->GetValue();
```

Namespaces

- A namespace encapsulates a set of functions, classes, and other namespaces
- If I have a really big piece of code, with several external pieces of code, what are the chances of two functions / classes having the same name somewhere in the code?
- You've already seen one namespace used a lot, without knowing it (std)

```cpp
// path/to/file.h

namespace path {
namespace to {

class MyClass {
}

void MyFn();

} // namespace to
} // namespace path
```

```cpp
// main.cpp
#include "path/to/file.h"

int main() {
  path::to::MyClass myClass;
  path::to::MyFn();
}
```

```cpp
// path/to/file.cpp

namespace path {
namespace to {

MyFn() {
}

} // namespace to
} // namespace path
```

The using keyword

- Several ways to use the using keyword
  - These are the two most important ways

Using declaration

```cpp
#include "path/to/file.h"

int main() {
  // Imports a single thing
  // into the current scope.
  using path::to::MyClass;
  MyClass myClass;
}
```

Type alias

```cpp
// Far cleaner than typedef, and the
// arguments are the right way around!
int main() {
  using intvec = std::vector<int>;
  intvec vec;

  C::iterator_t it;
}

class IteratorC {
}

class C {
  using iterator_t = IteratorC;
}
```

ADL

- Due to a feature of C++ called "Argument dependent lookup", use the using declaration for some specific functions
  - This is a complex topic to be discussed in later weeks
- This is only relevant when using non-standard types
- Use this for the following functions
  - std::swap
  - std::[cr]begin
  - std::[cr]end
  - std::empty
  - std::size
  - std::data

```cpp
#include "myclass.h"

using std::begin;
using std::end;
using std::swap;

int main() {
  std::vector<MyClass> vec{{}, {}};
  swap(vec[0], vec[1]);
  for (auto it = begin(vec); it != end(vec); ++it) {
    std::cout << *it << '\n';
  }
}
```

Argument dependent lookup

- When looking up an unqualified function, the compiler first looks at the namespace of the type of the arguments
  - If it contains a matching function declaration, it uses that
  - Otherwise, it falls back to the normal function lookup
- Use ADL for std::swap, std::begin, and std::end
  - You may write "using std::swap;"
  - Do not write "using namespace std;"

```cpp
// myclass.h

namespace ns {

void MyClass {
}

void swap(MyClass&, MyClass&);

}
```

```cpp
#include "myclass.h"

int main() {
  ns::MyClass v1, v2;
  std::swap(v1, v2);  // Calls std::swap
  {
    // Import std::swap to the current scope
    using std::swap;
    int i, j;
    swap(i, j); // calls std::swap
    swap(v1, v2); // calls ns::swap
  }
}
```