COMP6771
Advanced C++ Programming

3.2 – Class Types

What is
Object-Oriented Programming?

- 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 concepts:
  - 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

C++ Class Types

A **class type** in C++ is defined as one of the below three entities:

- `structs`
  - Same as they are in C.
  - A structure made up of data of any type.
- `unions`
  - Same as they are in C.
  - We will not be covering unions much in this course.
- `classes`
  - New in C++.
  - Classes are virtually identical to structs.

C++ Classes

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

- A class:
  - Defines a new type
  - Is created using the keyword `class`
  - May define some members (methods, data)
  - Contains zero or more public, protected, or 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.

Classes vs. Structs

- A class and a struct in C++ are almost exactly the same.
- The only technical difference is that:
  - All members of a struct are public by default
  - All members of a class are private by default
- Structs generally are used for simple types with few or no methods.
  - These kinds of structs are called PODs: plain old data.
- Intended use of your type should determine whether it is a struct or class

```
// default everything is private
class foo {
   int member_;
};

// default everything is public
// we often times denote this with
// a different style of member naming
// note the lack of a trailing _
struct footwo {
   int member;
}
```

Member Access Control

- This is how we support encapsulation and information hiding in C++.
- Can have multiple sections of the same **access specifier**.
- They are:
  - `public` – accessible by everyone
  - `protected` – accessible by children of the class and the class itself.
  - `private` - accessible only by this class
  - `virtual` - cover this in the polymorphism module.

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

// Accessible by this class & children.
// Will discuss more later in the course
protected:

// Accessible only by this class
private:
  void private_member_function();
  int private_data_member_;

// Can define the same access specifier again
public:
};
```

Constructor

- Constructors define how this object should be initialised.
- A constructor has the same name as the class and no return type.
- The constructor that is callable with zero arguments is called the **default constructor**.

```
#include <iostream>

class my_class {
public:
    // this is a constructor
    my_class(int i) {
        i_ = i;
    }
    get_val() {
        return i_;
    }

private:
    int i_;
};

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

Construction Order

Class construction follows this pseudocode algorithm.

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

Member Initialiser List

- The initialisation phase occurs before the body of the constructor is executed, regardless of whether a member initialiser list is supplied.
- You are able to provide initial values for data members before entering the constructor body.
- This avoids initialising a data member only to have its value overwritten.
- You should **always** use a member initialiser list.
- You also must ensure you initialise data members in declaration order.
  - Otherwise the behaviour is undefined.

```cpp
class user {
public:
  user(std::string name, int age)
  // everything following the colon
  // is the member initialiser list
  : name_{name}, age_{age} {
    /* more complex set-up */
  }

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

Delegating Constructor

- A constructor may call another constructor from the member initialiser list.
- Since the other constructor must construct all the data members, you should not specify anything else in the list.
- The other constructor is called completely before this one.
  - This means the lifetime of the object has begun!

```
class point2d {
public:
  // the default ctor delegates to the
  // 2-arg constructor, supplying
  // default values
  point2d() : point2d(0, 0) {}

  point2d(float x, float y)
  : x_{x}, y_{y} {}

private:
   float x_;
   float y_;
};
```

In-class Initialisers

- It can be easy to forget to initialise all data members, especially when there are many.
- You can use in-class initialisers as a last resort fallback if a data member is not initialised with a member initialiser list.
- It is best to minimise their use, as having initialisation logic spread out can be hard to read.

```
class point2d {
public:
  point2d(float x, float y)
  : x_{x}, y_{y} {}

private:
   // with no default constructor,
   // the in-class initialisers
   // will be used instead.

   float x_ = 0;
   float y_ = 0;
};
```

`explicit`

- If a constructor for a class is callable with a single paramater, the compiler will create an implicit type conversion from the parameter to the class type.
- This **may** be the behaviour you want
  - But probably not
- You have to opt-out of this implicit type conversion with the `explicit` keyword.

```
class point2d {
public:
  // single arg parameter
  explicit point2d(float n)
  : x_{n}, y_{n} {}

  // this ctor is also callable with a single arg
  explicit point2d(float *f, int n = 2)
  : x_{f[0]}, y_{n >= 2 ? f[1] : f[0]} {}

private:
  float x_;
  float y_;
};

point2d ex() {
   // error: single arg constructor is explicit
   return 3.0f

   float arr[2] = {0.0f};
   // error: 2-arg constructor also is explicit
   return arr;
}
```

Destructor

- Called when the object goes out of scope.
- How would you use one?
  - Freeing malloc'ed memory.
  - Closing files.
  - Unlocking mutexes.
  - Aborting database transactions.
  - Any kind of resource clean-up.
- `noexcept` is part of C++ exceptions (we will cover this later)

```
// from Java: a boxed integer
class integer {
public:
   integer(int i) : ptr_{new int{i}} {}

   // never forget to free a pointer
   // ever again!
   ~integer() {
      delete ptr_;
   }

private:
   int *ptr_;
}
```

Special Member Functions

- There are six special member functions:
  - The default constructor
  - The copy constructor
  - The move constructor
  - The destructor
  - The copy-assignment operator
  - The move-assignment operator

```
class foo {
public:
  // default constructor
  foo();

  // copy/move (cover later) constructor
  foo(const foo &other);
  foo(foo &&other);

  // destructor
  ~foo();

  // copy/move (cover later) operators
  foo &operator=(const foo &other);
  foo &operator(foo &&other);
};
```

Synthesised Special Members

- The compiler is able to *synthesise* definitions for the special member functions if the user does not provide one.
- The synthesised version:
  - Default construction: tries to default construct all the data members in turn
  - Copy construction: calls the copy constructor of each data member
  - Destruction: goes in *reverse-order*, destructing each data member.
  - Etc.
- It is possible to opt-*into* synthesis with `default`.
- Likewise, it is possible to opt-*out* of synthesis with `delete`

```cpp
class point2d {
public:
  // even one user-supplied ctor stops
  // the compiler-generated default.
  point2d(float a, float b);
  point2d() = default; // opt back into it

  // now this class is no longer copyable
  point2d(const point2d &other) = delete;

  // ensure compiler-synthesised dtor.
  ~point2d() = default;

private:
  float x_;
  float y_;
};
```

Removing Unneeded Special Members

- 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;"

Non-Static Members

- A class can have member functions (methods).
- A class can also have data members.
- The size of a class only depends on the types and declaration order of its members.
  - Due to alignment requirements, the compiler may insert padding into your class.
- `sizeof(class) >= 1` (why?)

```
class foo {
public:
  void speak();
private:
  int i;
  void *ptr;
  bool b;
};

class foo2 {
public:
  void speak();
private:
  bool b;
  int i;
  void *ptr;
};

static_assert(sizeof(foo) == 24); // why?
static_assert(sizeof(foo2) == 16); // why?
```

Incomplete Types

- An incomplete type is a type that has been declared but not defined.
- You can only form pointers and references to incomplete types.
  - Except `void&` is illegal.
- Because of this restriction, a class cannot have data members of its own type.
- Since a class is considered declared once its name has been seen, it can have pointer/reference members to 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;

  // this, however, is fine.
  node *next;

  // this is fine, too.
  node &next;
};
```

The `this` Pointer

- Member functions have 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
  - As of C++23, you may now define it too.
- 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 `this->` to identify them
- It is possible to overload a method based on the constness of `this`.

```
class point2d {
public:
  point2d(float x, float y) : x_{x}, y_{y} {}

  float x() {
  // the signature of this method is really
  // float(point2d * const this)
    return x_;
  }

  float x() const {
  // the signature of this method is really
  // float(const point2d * const this)
    return x_;
  }
private:
   float x_ = 0;
   float y_ = 0;
};
```

`const` Objects

- Member functions are by default only callable by non-const objects.
- You can declare a const member function which is callable by both const and non-const objects
- A const member function may only modify `mutable` members
    - There are **very few** use cases for mutable
        - One example is as a cache
    - A mutable member *should* mean that the physical state of the member can change without the logical state of the object changing.
    - In practice, this isn't always the case.

```
class point2i {
public:
  point2d(int a, int b) : x_{a}, y_{b} {}

  const int& x() const { return x_; }
  int &x() { return x_; }

  const int& y() const { return y_; }
  int &y() { return y_; }

private:
  int x_;
  int y_;
}

const auto p = point2i{1, 2};

p.x(); // OK! const-qualified method called
p.y() = 4; // error: calls a non-const method
```

Static Members

- Static members belong to the class, as opposed to any particular object.
- Static methods are callable without any instance.
- Static methods are never const-qualified.
- Static data members' lifetime ends when the program ends.
- Use static members when something is associated with a class, but not a particular instance.

```
class point2d {
public:
  static point2d make_point(float a, float b) {
    return point2d(a, b);
  }

  ~point2d() {
    n_live_ -= 1;
  }

private:
  point2d(float a, float b) : x_{a}, y_{b} {
    n_live_ += 1;
  }

  inline static int n_live_ = 0; // since C++17

  float x_;
  float y_;
};
```

Friends

- A class may declare another function or class as a `friend`.
  - Note: this does not declare the friend class or function itself.
  - It only says *if* such a class or function exists, then it is a friend.
- Friends are able to access the private members of the class.
- For a class `C` and friend `F`:
  - `F` is not a friend of the children of `C`.
  - `F` is not a friend of classes `C` is a friend of.
  - `F` is not a friend of any parent classes of `C`.
- Friends are *always* public.

```
class point2d {
public:
  /* Other implementation details... */
  // declare distance as a friend of point2d
  friend float dist(point2d &, point2d &);

private:
  float x_;
  float y_;
}

float dist(point2d &l, point2d &r) {
  // because dist is a friend, it can access
  // the private members of point2d
  return std::sqrt(l.x_ * r.x_ + l.y_ * r.y_);
};
```

Hidden Friends

- It is possible to declare and define a friend inline in a class.
- This “hidden” friend has different look-up rules than usual:
  - Only discoverable through ADL
  - With a separate declaration outside the class, they are discoverable without ADL.
- Mostly used with operator overloading.

```cpp
namespace hf {
  class point2d {
  public:
    // other implementation details...

    friend float dist(point2d &l, point2d &r) {
      return std::sqrt(l.x_ * r.x_ + l.y_ * r.y_);
    }

  private:
    float x_, y_;
  };

  float dist(point2d &, point2d &);
}

void ex() {
  hf::point2d p = {1, 2}, q = {3, 4};

  // OK: hidden friend found through ADL
  dist(p, q);

  // OK: found through 2nd declaration outside class
  hf::dist(p, q);
}
```

The Power of Friendship

Friendship seems to break encapsulation. Why use it?

- In general, friendship should be used sparingly.
- Friendship can be used between two interconnected classes.
  - E.g., a container and its iterator.
- Friendship can also be used for **API Extension.**
  - This does not mean unrelated code can extend an API.
  - It is for a different calling style for a logical operation of a type.
  - E.g., for a `vec3` class, the `cross()` operation should be a friend since `cross(a, b)` makes more sense than `a.cross(b)`.
- For certain operator overloads, it is more convenient for them to be defined as non-member friend functions.

Interface & Implementation

- We are used to declaring functions in header files and defining them in .cpp files.
- With classes we have two options:
  - Define the class completely in the header.
  - Define the class in the header but only declare methods. Define methods in the .cpp file.
- Defining everything in the header is easier for us but can slow down compilation due to duplicate code.
- Either way, static data members must be defined in a .cpp file unless declared `inline`.

```
// point.hpp – interface file
class point2d {
public:
  point2d(float a = 0, float b = 0);

  float &x() { return x_; }

private:
  static int cnt;
  float x_;
  float y_;
};

// point.cpp – implementation file
int point2d::cnt = 0; // static member

point2d::point2d(float a, float b)
: x_{a}, y_{b} {}     // ctor implementation
```

Feedback (stop recording)