4 - Class Definitions

January 13, 2025

COMP2404

Darryl Hill

1. Imperative to Object Oriented
2. Access Specifiers
3. Class Members
4. Code Organization
5. Variable Scope
6. Namespaces

Imperative to Object Oriented

C++ was once known as C with **classes**.

- Classes provide an ***Object Oriented*** way of organizing data and functions related to that data.

We will start by writing the code ***Imperatively***

- C uses ***imperative*** style.
- We will convert it to ***Object Oriented*** code.

Along the way we will show:

- How **classes** and ***Object Oriented*** programming provide a useful way to organize code.
- How to allocate **classes** dynamically

University System

Let's say we want a **University** system

- It will have information about all the `Students` there
- Each `Student` will have a **name**, **student number, major, gpa**.
- We will want to do operations such as
  - print all `Students`, or
  - print all passing `Students`.

We will use arrays to store the `Students` information (simulating a database).

coding example <p1>

University System

We've stored all the information in separate arrays, but it would make sense to do something more convenient

- such as have them all together in memory.

In C we could use a **struct**.

- C++ also has **structs**.

In C++ **structs** and **classes** are the ***same***

- except for the access modifiers.
- We will use ***classes*** in C++.

We will make a `Student` **class** to store all information related to students.

coding example <p2>

University System

Instead of printing all the information separately, it makes sense to have a `print` function for `Students`.

We can make the **print** function a part of the `Student` class

- That way information and the functions that act on them appear together.
- This also gives the functions access to *private* member variables.

coding example <p3>

University System

We are still initializing `Students` in a primitive way.

- C++ gives you a special ***constructor*** function to handle this.
- Since it is a function, it has access to private members.

Now we can ***hide*** or ***encapsulate*** the information.

- `Student` data can only be modified in ways we approve of.

We can also allocate `Students` dynamically if we wish.

- We only allocate memory as we need it.
- We can delete it when we are done.

coding example <p4>

Imperative to Object Oriented

We have shown how **classes** and ***Object Oriented*** programming is a natural way to organize programs.

- The overhead compared to imperative programming is still relatively low.

We have also begun to demonstrate ***encapsulation***.

- We don't give access to information in unnecessary ways.
- Only functions belonging to the **class** had access to ***private*** member variables.

Next we will cover **classes** in more detail.

Access Specifiers

Industrial grade software is typically huge

- Millions of lines of code
- Hundreds of libraries and packages
- Must protect your code from unauthorized changes

Restrict access to unnecessary details, information, access whenever possible

- **Principle of Least Privilege**
- everything should be as close to private as possible
- example – you only want qualified people to access the inner workings of your car

Of course, some (`public`) access is necessary

Class Access Specifiers

Public access

- class member is visible to all objects and global functions

Protected access

- class member is visible by subclasses only
- different from Java
- make use of this only when using inheritance

Private access

- not visible to other classes or global functions
- visible to other members of the **same class**
  - different from Java

Classes in C++

Class Definitions

- `class` keyword followed by class name followed by braces `{}`, followed by semi-colon `;`

In the `class` block are the `class` members

- data members (instance variables)
- member functions (methods)

We can use access specifiers to partition members by access level

- `public`, `private`, `protected`
- default access level for classes is `private`

Classes in C++


```
class Student{

    public:  // start of the public members

        // some functions and/or variables

    private: // start of the private members

        //some functions and/or variables
};
```

Objects in Memory


```
Student casey("100999999","Casey");
Student joe;
```

Observe there is no “new” here. How did we make new objects?

- C++ gives you the option of *statically* or *dynamically* allocating objects.
- If you statically allocate an object as a local variable in a function, then
  - Object is allocated to the **Stack**.
  - Access is fast.
  - When the **stackframe** is popped off, the object is deleted for you.
  - Convenient for temporary objects.

Objects in Memory


```
Student* casey = new Student("100999999","Casey");
Student* joe = new Student;
```

We can also use "new" to allocate objects on the **Heap**.

- The **Heap** is slower to allocate objects,
  - but the objects last as long as we like
- These objects are never deleted until we call "delete"
  - We should consider when to do this in our design.
- More on this later

Member Function Implementations

A Class in C++ should consist of 2 files:

- A header file (using the `.h` extension) contains the class definition.
  - Data member declarations.

    ```
    string name;
    string id;
    ```
    etc.

  - Member function prototypes (usually not the code!)

    ```
    void print();
    bool isPassing();
    ```

- A source file (using the `.cc` extension)
  - This contains the member function implementation (the actual code)
  - Static data member initializations
  - Related global functions

Member Function Implementations

A function implementation is the code for a function.

- Written in the .cc file.
- Has scope resolution operator, the function definition, and the code.

```
void Student::print(){
     cout<<"Name: "<<name<<endl;
     cout<<"Id:   "<<id<<endl;
}
```

Member Function Implementations

For very small programs, we may include implementation in the class definition

- Getters for example
- This gets messy quickly – discouraged for larger programs

For all other programs, the implementation should be separate

- There are a couple of reasons:
  - Principle of Least Privilege
  - Can help avoid circular `#includes` between your files

Standard Member Functions: Constructors

Job of a constructor to ***initialize all data members***

- There are a few ways to initialize data members
- We will start with the most conventional
  - assignment operators

Constructor with no arguments – **default constructor**

- can be very important for things like arrays
- we decide if there is a default constructor

C++ classes can have multiple constructors

- cannot call "super" as with Java to make use of multiple constructors
- C++ can leverage other constructors, but uses a different syntax

Standard Member Functions: Constructors

Prototype (in the header file):

```
class Student{
    public:
        Student();
        Student(string, string);
};
```

Implementation (in source file):

```
Student::Student(){
    name = "unknown";
    id   = "unknown";
}
```

Other Standard Functions

There is no `toString()` function in C++

- initially we will use `print()` functions
- later we will see how to overload the stream insertion operator `<<`
  - C++ equivalent to `toString()`

Example <**p4**> from previous slides is bad – body of functions are within the class definition.

- against C++ conventions (though the compiler is fine with it)
- Keeping headers and source separate has actual advantages:
  - Compiler only needs the function prototype (header) to report usage errors
  - Header can be `#include`-d, and the source still compiled separately and linked.
  - Can help mitigate circular references.

programming example <p1>

Allocating Classes

Like primitive variables, we have choice in how to allocate classes.

- Statically allocated means memory is allocated where it is declared.
- Dynamically allocated means allocated on the Heap.

There is an extra complication using **classes** with **arrays**.

For now we will ***statically allocate*** arrays, but we still have 2 choices in the type of array:

- Array of objects, or
- array of object ***pointers***.
- Arrays of **objects** automatically call the **default constructor** of the **class**.

programming example <p2>

Compiler

The C++ compiler is (deliberately) primitive

- It does not organize the code for you (like Java)

Consider the command: `g++ -c Date.cc`

- The compiler starts reading `Date.cc`.
- When it sees `#include "Date.h"` it jumps to that file and continues reading.
- Think of all `#include`-d files as one large file, in order.
  - If the compiler reads it top to bottom, will it make sense?
  - What information does the compiler need to ensure no errors?
  - What are the potential pitfalls?

Class Interface

A **Class Interface** is *not* the same as a Java interface.

The **Interface** refers to the part of the class that can be accessed and used by other classes. It consists of

- class name and
- public members.

This defines how the user interacts with your class.

In C++, this is contained in the header file.

- Though there are also private and protected members in the header.

Class Interface

To use a class, you `#include` the header file.

- This contains the public interface.
- This is enough information for the compiler to ensure that you use the class correctly.
- It is exactly like forward referencing your functions.

To make an executable using this class, you then must link your code to the object code.

- I.e., the implementation.

Users of your class do not need to see the source code.

- The interface and documentation give enough information for other developers to use your class.

Code Organization

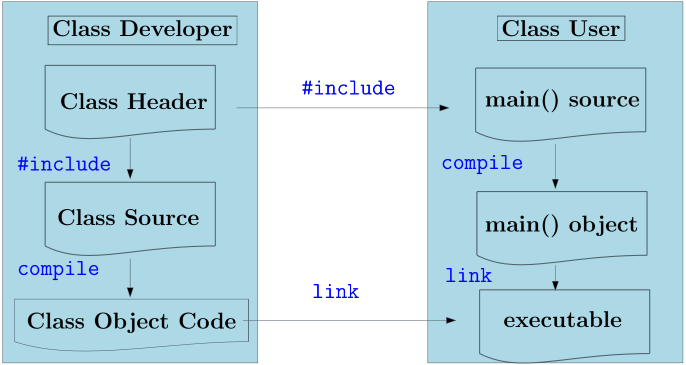
Who will be using your class?

- Mostly other developers
- Even when writing end user code, you may want API's or other developers to contribute.

Other developers need to know:

- Class name,
- public members,
- descriptions of functions where appropriate, and
- (sometimes) protected members.

Class Interface



- If we change our source code, we can recompile this separately from applications that use it.
- User only needs to re-link this source, not re-compile.

Class Header File

Make sure to include the ***header guards***.

- These guard against multiple `#include`-s.

`#include`-ing source code is inefficient.

- Unnecessarily forces recompilation of all code.
- This is frowned upon…mostly.
- **Source** file should `#include` the **header**, not vice-versa.

Understand what belongs in the header file (essentially your class API) vs the source file

- All source files should be compiled separately.
- Class users will only have to **relink** your code (for bug fixes, etc), which is fast.

Source Files

Source files contain:

- All class related source code
  - (i.e., member function implementations).

- By default, all functions you write are global!
- Use the **scope resolution operator** of the ***class*** to inform the compiler of what is a class member.
  - This lets the compiler resolve the visibility of variables and other functions from the class definition (`public, protected, private`).
  - Or to identify static data member initializations...
    - ...more on this later.

Variable Scope

Variable scope refers to where in the program a variable is visible.

- Block scope
  - including Class blocks.
- File scope
  - is outside of any block.
- Function scope for labels (we don't cover labels).
- Function prototype scope for parameters (we won't cover this either).
  - `int foo(int n, int x[n])`

Block Scope

A variable declared within a block has **block scope**.

- Visible within that block and all inner blocks (unless shadowed).
- Once we exit the block, the variable disappears and its value is discarded.
- Variables in inner blocks can hide variables in outer blocks.
  - Shadowing.
- If variables in nested blocks have the same name, the innermost block variable is the one used.
  - Try to avoid this - use unique identifiers when possible.
- Can always use the unary resolution operator to access a global value.
  - Other shadowed variables remain invisible, unless blocks are ***namespaces***.
  - Coming soon.

File Scope

A variable in File scope:

- Is declared outside of all blocks
- Visible everywhere in that file.
  - Global variables, or
  - global functions.

Such a variable can be accessed from another file using the `extern` keyword.

- Without `extern` the compiler will think it is a new variable declaration.

Coding example <p3>

Namespaces

What is a **Namespace**?

- Not a class! Not a package! Has properties of both.
- Closest equivalent is Java ***package***, but more flexible.
- It is the definition of a (named) scope.

To use an element from a **namespace** you must scope it in by either

- using the `using` keyword
- use the ***scope resolution operator*** `::`

Namespaces

A **namespace** may be unnamed, then it is automatically scoped in.

- Has only ***internal*** linkage.
- Visible only to the current ***translation unit***
  - Current source and #include-d headers.
  - I.e., everything that makes this object file.

This is like a global variable, but only for a select group of files.

programming example <p4> - see Makefile for ***variables***.