3 - Programming Conventions

September 15, 2022

COMP2404

Darryl Hill

Programming Conventions

Every place you work will have its own conventions.

- Makes code easier to read.
- Makes interfaces consistent.
  - If I expect a constant reference, that is what I should find.
- Helps your code communicate with the code from the rest of the team.

Some naming conventions are universal.

- They do not create syntax errors, but must be observed.
- They are a communication tool.
  - Makes your code easier to understand.
  - Going against them would confuse other programmers.

Naming Conventions

Constant names are all UPPERCASE:

- Example: MAXARRAY

May also separate words with an underscore:

- MAX_ARRAY

Variables begin with lower case letter.

Use **camelCase** or **underscores** for multiple words.

- **numElements** or **num_elements**

Naming Conventions

Names should be self-documenting without being overly long.

- If you are storing a name, then the variable should be `name`, not `n`.
- Can use the scope as context – instead of `studentId`, if it is in the `Student` class just use `id`.

The class name provides the additional information.

Single lower case letters are commonly used for looping – `i` for index, but also `j`, `k`, `l`, etc.

Naming Conventions

Data types or **Classes** begin with upper case

- (except for primitives and **strings**).
- `Date`, `Student`, etc.
- `CamelCase` or `Under_Scores` can be used if it is a compound name (less common).

Functions:

- Begin with a lower case letter:
  - `sin()`, `help()`
- Can use `camelCase()` or `under_scores()` for compound names.

Aim for names that are short and descriptive.

Indentation

White space before the first character of the line.

- Promotes readability
- Usually used to denote blocks
- {} braces tell the compiler where the blocks are,
- indentation tells other programmers where the blocks are.

```
if (bool){
    // indented lines of code
    // to indicate a block
    // (that is, statements within
    // curly braces {})
}
```

Blocks

Block:

- Sequence of statements between a pair of braces `{}`.

Nested block:

- Block within another block.
- Should have extra indentation.

Nesting level of a block:

- The number of open and not yet closed braces before a block.
- Depth from file scope.

Indentation Rules for Blocks

- All statements within a block should have the same (indentical!) indentation.
- All blocks at the same nesting level should have identical indentation.

Indentation Rules

There are three acceptable styles of brace indentation.

- This is for consistency of communication between programmers.

```
if (bool){
    //some stuff
}
```

```
if (bool)
{
    //some stuff
}
```

```
if (bool){ //one line of stuff }
```

Indentation Rules

How much should we indent?

Convention:

- 2 spaces.
- 4 spaces.
- One tab (not always portable between editors).

Consistency is most important.

Comment Blocks

Blocks of comments are used to:

- Explain the program,
- explain a class, or
- explain a complex or critical block or section of code.

Block of comments in the main function specify

- the purpose of program,
- how to use it,
- the authors, and
- all revisions.

Comments

Each class should have a block of comments before the class definition describing

- the purpose of the class, or
- any functions or variables that require additional information.

Inline comments:

- Should be short – one line, or part of a line, or two short lines.

```
int x = input(); // input from keyboard

// this expression finds the intersection of two lines in
// the form m = ax+b
intersect = <some expression>
```

Comments

Larger blocks of comments are used to

- explain a class, or
- explain a section of code that might be confusing.

```
/*
*   If I need a paragraph to explain something,
*   I can use this style of comments.
*   The * to begin each line is cosmetic, but encouraged.
*   When we are finished we end with
*/

<some code to be explained>
```

Other Conventions

- **Don't** use **structs** when you should use **classes**..
- **Don't** use global variables.
- **Don't** use too many global functions
  - In some cases they are needed, because C++.
- **Don't** pass objects (classes) by value.
  - This makes copies, it is slow and wasteful of memory.
- **DO** use return parameters instead of return values (in this class) for all but the simplest return values.
- **DO** reuse code whenever possible.
  - Don't copy and paste code over and over – if you are doing this it is time to ***refactor***.
- **DO** perform basic error checking
  - sanitize any data that goes into your class.

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 gradually 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`.

Since there are many `Students` we will use arrays to store the information.

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.