CO2402 Advanced Programming with C++

Style Guide

Introduction

The following style guide is primarily intended for CO2402: Advanced Programming with C++.

You are required to use this style guide for code submissions for this module. You may be required to use it for other modules as well. The aim of this guide is to make your code style consistent and readable.

Please note that this style guide is much smaller than a real-world guide would be!

How should you name things?

- Absolutely no *meaningless* names! The names of variables and functions should make it easy to understand what the code is doing. Meaningless names are evil!
- A function must have a strong verb (a "doing word"). The form should be *verb* followed by an *object*. The object is often a noun (a "name"), e.g.

```
UpdateRecord();
HarvestPumpkin();
```

  - If a function returns a value then use a description of what is returned in the name, e.g.

```
GetName(); // returns the name
```

- A variable name is likely to be a noun, e.g.

```
Record studentRecords;
```

- A name should be neither too short nor too long…
  - Short variable names can only be used if the context is completely unambiguous, e.g. x and y are acceptable in the case of coordinates.
  - Single character variable names are usually only okay for loop counters and maths.
  - If you do use a single character for a loop counter then it must follow the convention: i, j, k, unless there is an explicit reason otherwise, e.g.

```
for( int i = 0; i < total; i++ )
```

  - Don't abbreviate by just removing one character from a word – what's the point?
  - Abbreviations must be consistent.

Naming Conventions

- Underscores should **not** be used. If a name is made up of more than one word then each succeeding word should begin with a capital letter, e.g.

```
largestValue;
```

- **Variable names** begin with a lower-case character. If a variable is made up of more than one word then each word must begin with an upper-case character (apart from the one beginning the variable name). All other characters must be lower-case. For example:

```
float maximum;

int numberOfRecords;

Employee fullTimeEmployees;
```

- **Function names** begin with an upper-case character. Each word must begin with an upper-case character. All other characters must be lower-case. For example:

```
DisplayRecord( );
```

- **Constants** begin a lower case *k*. Each word must begin with an upper-case character. All other characters must be lower-case. For example:., e.g.

```
kMaxPumpkinSize
```

- **Member variables** of classes and structs begin a lower case *m*. Each word must begin with an upper-case character. All other characters must be lower-case. For example:

```
mProductCode
```

- **Global variables** begin a lower case *g*. Each word must begin with an upper-case character. All other characters must be lower-case. For example:

```
gDatabaseSize
```

- **Pointers** begin a lower case *p*. Each word must begin with an upper-case character. All other characters must be lower-case. The *m* or *g* prefix is placed *before* the *p* prefix, e.g.

```
pDatabaseRecords

mpDatabaseSize

gpDataFormat
```

Here are the prefixes in table format. The first character of a name indicates its type.

Type	**Prefix**	**Example**
constant	k	kMaxSize
global variable	g	g
pointer	p	pElement
member variable	m	mSize
Member variable which is a pointer	mp	mpData

- **Types (**declared as **Classes, Structs, Enums, Typedefs and Using)** begin with an upper-case character. Each word must begin with an upper-case character. All other characters must be lower-case. For example:

```
Timer

FullTimeEmployee
```

Variables

- All variables should be given a default (and sensible) value when they are declared if this is at all possible. Uninitialised variables are evil!
- A variable should only be used for one purpose only.
- A pointer should be assigned a value of `nullptr` if is not currently pointing at a known memory address. Do not use 0 or NULL – they are evil!
- Each variable should be declared on its own line, with its own type preceding it. Multiple declarations on the same line are evil!

```
int age;

string name;
```

- Pointers are declared with asterisk next to the type. Think of the type as "pointer to a …"

```
CQueue* pProductList = new CQueue;
```

- Pre-processor Macros (as in #define) should be avoided if at all possible – they are evil! ALWAYS use const in preference where values are concerned. If you must use them (e.g. for conditional compilation), then they should be named in ALL_CAPITALS

Layout and code organisation

- Use a tab character rather than spaces.
- Tab distance should be 4 (i.e. the equivalent of 4 spaces).
- Generally speaking put spaces around all mathematical operators and most other operators, e.g.

```
result = myMoney + yourMoney;

bill && ben
```

- It is difficult to be more precise than "most other operators" without a long list, but unary operators such as increment and decrement don't get spaces whilst a logical operator would, e.g.

```
daysGoneBy++;

christopher || alice
```

- The curly braces of a control block are aligned with the left hand side of the block. Each brace is placed line of its own.

```
while( currentSpeed < maxTolerance )
{
    IncreaseThrust( );
}
```

- Avoid excessively long lines of code and comments. If it goes off the side of the editor window, it is too long!
- Each class should be defined in its own header file. Method definitions should be in an associated source file.

Global Variables

- In general you should avoid the use of global variables. It is better to use access routines in the place of global data.
- There are, however, some valid reasons to use global variables. These include:
    - Declaring constants.
    - Eliminating tramp data – a variable that is used in many functions and needs to be passed on and on via a function chain. It will be obvious when you see this happening! The reason for this is that the data is used throughout a program. This genuinely would appear to be global data.

Comments

- Write comments at the level of the code's intent.
    - Focus on **why** rather than **how**.
- Place a comment before each block of statements. Short comments to the right of a line of code are okay, but NEVER place a comment *below* the code to which it pertains.
- Document the source of algorithms.
- Every method needs a comment section which briefly but clearly describes what the method does. The comment section can be placed in the header file or in the source file.

```
/** @brief function short description
 *
 * function longer description if need
 * @param[in]  param1_name  description of parameter1
 * @param[in]  param2_name  description of parameter2
 * @return return_name return description
 */
```