**CS1D Project 2 – Big O(Richard)**

**Coding Standards**

**Class Names**

* Use upper case letters as word separators, lower case for the rest of a word
* First character in a name is uppercase
* No underbars ('\_')

**Example**

class NameOneTwo

class Name

**Class Names**

* Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
* Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.

**Method Names**

* Use the same rule as for class names.

**Example**

class NameOneTwo  
{  
public:  
 int DoIt();  
 void HandleError();  
}

**Method Names**

* Usually every method and function performs an action, so the name should make clear what it does: GetStadiumName() instead of GetTeamName(), SetStadiumName(). This will also make functions and data objects more distinguishable.
* Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.
* Prefixes are sometimes useful:
  + *Is* - to ask a question about something. Whenever someone sees *Is* they will know it's a question.
  + *Get* - get a value.
  + *Set* - set a value.
* For example: GetStadiumName()

**QT Items**

* When QT is used as a GUI the following naming conventions must be used

|  |  |
| --- | --- |
| QT Buttons | btn\_*Brief\_Descrition* |
| QT Line Edits | LE\_*Brief\_Descrition* |
| QT Tables Widget / Tableview | tbl\_*Brief\_Descrition* |
| QT Combo Boxes | CB\_*Brief\_Descrition* |
| QT Spin Boxes | SB\_*Brief\_Descrition* |
| QT Date Edit | DE\_*Breif\_Descrition* |

**C++ Function Names**

* In a C++ project there should be very few C++ functions.
* For C++ functions use the GNU convention of all lower case letters with '\_' as the word delimiter.

**Example**

int some\_bloody\_function(){

. . .   
 }

**Pointer Variables**

* pointers should follow normal naming conventions where the first word of the variable name is all lowercase followed by the next word in uppercase (Camelcase)
* place the *\** close to pointer type

**Example**

String\* name= new String;

**Static Variables**

* Static variables should follow normal naming conventions where the first word of the variable name is all lowercase followed by the next word in uppercase (Camelcase)

**Example**

class Test  
 {  
 public:  
 static StatusInfo status;  
 }

**Required Methods for a Class**

Following Methods are a MUST have

* **Default Constructor**
  + If the default constructor is sufficient add a comment indicating that the compiler-generated version will be used.
  + If your default constructor has one or more optional arguments, add a comment indicating that it still functions as the default constructor.
* **Virtual Destructor**
  + If your class is intended to be derived from by other classes then make the destructor virtual.
* **Copy Constructor**
  + If your class is copyable, either define a copy constructor and assignment operator or add a comment indicating that the compiler-generated versions will be used.
  + If your class objects should not be copied, make the copy constructor and assignment operator private and don't define bodies for them. If you don't know whether the class objects should be copyable, then assume not unless and until the copy operations are needed.
* **Assignment Operator**
  + If your class is assignable, either define a assignment operator or add a comment indicating that the compiler-generated versions will be used.
  + If your objects should not be assigned, make the assignment operator private and don't define bodies for them. If you don't know whether the class objects should be assignable, then assume not.

**The Law of The Big Three**

A class with any of (destructor, assignment operator, copy constructor) generally needs all 3.

**Example**

An example using default values:

class Planet{  
public:  
 // The following is the default constructor if  
 // no arguments are supplied:  
 Planet(int radius= 5);  
   
 // Use compiler-generated copy constructor, assignment, and destructor.  
 // Planet(const Planet&);  
 // Planet& operator=(const Planet&);  
 // ~Planet();  
};

**Braces *{}* Policy**

**Brace Placement**

* if (condition)

{

. . .

}

* while (condition)

{

. . .

}

**When Braces are Needed**

All if, while and do statements must either have braces

**Always Uses Braces Form**

All if, while and do statements require braces

if (somevalue == 1)

{  
 someValue = 2;  
}

**Add Comments to Braces**

Adding a comment to the start of any condition statement

// Comment Goes Here Describing

// What Will Happen If Necessary

while (condition)

{

. . .

}

**Parenthesis *()* with Keywords and Functions Policy**

* Do not put parens next to keywords. Put a space between..
* Do not use parens in return statements when it's not necessary.

Example

* if (condition)

{

. . .

}

* while (condition)

{

. . .

}

setStadiumName(“Cool Stadium Name”);

return capacity;

**If Then Else Formatting**

**Layout**

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

* if (condition)

{

. . .

}

else if (condition)

{

. . .

}

else

{

. . .

}

**Condition Format**

Always put the constant on the right hand side of an equality/inequality comparison. For example:

if (errorNum == 6) …

**Naming Class Files**

**Class Definition in One File**

Each class definition should be in its own file where each file is named directly after the class's name:

ClassName.h

**Implementation in One File**

In general each class should be implemented in one source file:

ClassName.cpp

**But When it Gets Really Big...**

If the source file gets too large or you want to avoid compiling templates all the time then add additional files named according to the following rule:

ClassName\_Section.h

**\* Section** is some name that identifies why the code is chunked together. The class name and section name are separated by '\_'.

**Ordering is: public, protected, private**

Notice that the public interface is placed first in the class, protected next, and private last.

It makes sense then to have the interface first.

**LIFECYCLE**

The lifecycle section is for methods that control the lifecycle of an object. Typically these methods include constructors, destructors, and state machine methods.

**OPERATORS**

Place all operators in this section.

**OPERATIONS**

Place the bulk of a class's non access and inquiry method methods here. A programmer will look here for the meat of a class's interface.

**ACCESS**

Place attribute accessors here.

**INQUIRY**

These are the *Is\** methods. Whenever you have a question to ask about an object it can be asked via in *Is* method. For example: IsOpen() will indicate if the object is open. A good strategy is instead of making a lot of access methods you can turn them around to be questions about the object thus reducing the exposure of internal structure.

**What should go in public/protected/private?**

**Public Section**

Only put an object's interface in the public section. **DO NOT** expose any private data items in the public section. At least encapsulate access via access methods. Ideally your method interface should make most access methods unnecessary. Do not put data in the public interface.

**Protected and Private Section**

Keeping everything all private seems the easiest approach. By making the public methods virtual flexibility is preserved.

**Use Header File Guards**

Include files should protect against multiple inclusion through the use of macros that "guard" the files.

**When Not Using Namespaces**

#ifndef FILENAME\_H  
#define FILENAME\_H  
  
#endif

The new line after the endif if is required by some compilers.

**When Using Namespaces**

If namespaces are used then to be completely safe:

#ifndef NAMESPACE\_FILENAME\_H  
#define NAMESPACE\_FILENAME\_H  
  
#endif

1. Replace *FILENAME* with the name of the file being guarded. This should usually be the name of class contained in the file.
2. When the include file is not for a class then the file name should be used as the guard name.
3. Historically many compilers require a new line after last endif.

**Different Accessor Styles**

**Why Accessors?**

Access methods provide access to the physical or logical attributes of an object. Accessing an object's attributes directly as we do for C structures is greatly discouraged in C++. We disallow direct access to attributes to break dependencies, the reason we do most things. Directly accessing an attribute exposes implementation details about the object.

If any of the above changed code would break. An object makes a contract with the user to provide access to a particular attribute; it should not promise how it gets those attributes. Accessing a physical attribute makes such a promise.

**Implementing Accessors**

Get/Set

* class X

{

public:

int GetAge() const

{

return aAge;

}

void SetAge(int age)

{

this->age = age;

}

private:

int age;

};