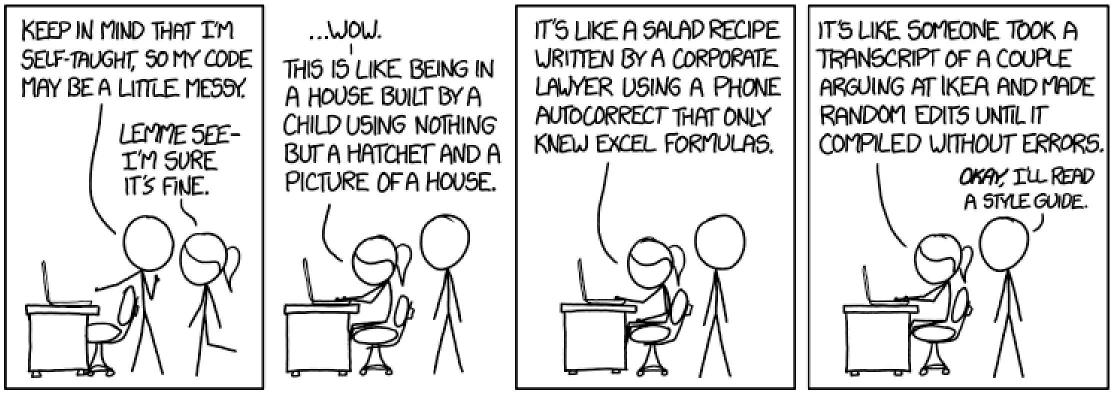
Lab 10 - C++ Coding Style Guidelines

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

Spring 2020

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Overview and Objectives

This document describes a set of C++ coding style suggestions based on practices commonly found in the body of code comprising the MOOS-IvP Open Source autonomy project. Some of the topics are by nature subjective, while some represent coding practices we advocate for more vigorously especially if you are a student of 2.680 or writing code to be included in the MOOS-IvP distribution.

This document covers *style* conventions or guidelines as opposed to its sister document which covers *structure* conventions. Both are part of the same discussion, but the latter is perhaps less contentious, and represents longer standing practices concerning the assurance of code correctness. While the former is a bit more subjective and addresses code *readability*.

- Put Your Name On Your Code
- Class Member Variables Use Snake Case Notation
- Class Functions Use Camel-Hump Notation
- Column Width
- Code Indentation Two Spaces
- Close Braces Should Be On Their Own Line
- Open Braces - When They Should Be On Their Own Line
- Commenting Class Functions
- Class Interface Organization
- Class Names - Short and Unique
- Function Return Statements Use Parentheses
- The main.cpp File and Structure

More C++ Structure/Style Resources

For some of the topics covered here, related discussions can be found on the web. Links to some of these are given in the related section and a few general references are further provided below. Books:

- *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Herb Sutter, Andrei Alexandrescu, 2004.
- *Safe C++: How to avoid common mistakes*, Vladimir Kushnir, 2012.

Web sites:

- The googlecode.com web site has a style guide that's worth checking out for comparison. http://google-styleguide.googlecode.com/svn/trunk/cppguide.html
- Stroustrup also provides a short discussion on coding standards and a few pointers of their own to other resources here: http://www.stroustrup.com/bs_faq2.html#coding-standard

2 Put Your Name On Your Code

Even if you think a piece of code is just a temporary hack that no one but you will ever use, put your name on it - *day one!* It is far too common to open a piece of software in a group project repository, and no one has any idea who wrote it.

Put your name at the top of every source code file. This includes all `.cpp` and header, `.h`, files, including `main.cpp` and the `CMakeLists.txt` file. A header block like the one below is strongly recommended, including name, organization, file name, and date created.

```
/*****************************************************************/
/*    NAME: Harry Chapin                                         */
/*    ORGN: Dept of Mechanical Eng, MIT Cambridge MA             */
/*    FILE: Widget.cpp                                           */
/*    DATE: Oct 12th 2004                                        */
/*****************************************************************/
```

If a major revision is implemented that either significantly changes the code function or was done by someone other than the original author, something like the below is appropriate:

```
/*****************************************************************/
/*    NAME: Ella Sudbury (originally by Harry Chapin)            */
/*    ORGN: Dept of Mechanical Eng, MIT Cambridge MA             */
/*    FILE: Widget.cpp                                           */
/*    DATE: Oct 12th 2004                                        */
/*    DATE: Jun 18th 2012 Multithreading added by E. Sudbury     */
/*****************************************************************/
```

3 Variables Use Snake Case Notation

Variables should begin with a lowercase letter. Even better if lower case variables are used exclusively. Variables with multiple word components should be connected with underscore character. This is often called snake_case notation. Some examples:

- `a`, `i`, `j`
- `one`, `result`, `var23`
- `m_verbose`, `initial_temperature`, `best_so_far`

4 Class Functions Use Camel-Hump Notation

Camel-hump notation uses a capital letter for each word component in the class name except for the first letter. By always beginning with a small letter, class functions are easily distinguishable from class names, which also use camel-hump notation. Some examples:

- `getFirstElement()`
- `addIvPFunction()`
- `solveIPP()`

5 Column Width

A column of code should be no greater than 80-85 characters wide. Limiting the column width is motivated by wanting to see more than one column on the screen and avoiding wrap-around in a text editor. Seeing code side-by-side is really useful, especially in C++ where much of coding is done on a class implementation and it's nice to see the class definition simultaneously. Most editors with a C++ mode will automatically elegantly indent a single line of code broken up over more then one line.

In Emacs you can set the fill column width in your `.emacs` file with:

```
;; Set the fill column width - used in auto-filling
(setq fill-column 80)
```

6 Code Indentation Two Spaces

The choice of either two or four spaces is simply a matter of style and motivated by wanting to have thinner columns of code, usually to get more than one column of code viewable with limited screen real estate. You may have to configure your editor to do this, as most editors have automatic indentation. Make sure your editor is not configured to have a TAB indent, which may map to 2 or 4 spaces. Try to make it explicitly two spaces.

Here's what it looks like with two spaces:

```
void handle(int a, int b)
{
  for(int i=0; i<a; i++)
    for(int j=0; j<b; j++)
      if(i > 100)
        if(j > 100)
          cout << "hello" << endl;
}
```

And here's what it looks like with four spaces:

```
void handle(int a, int b)
{
    for(int i=0; i<a; i++)
        for(int j=0; j<b; j++)
            if(i > 100)
                if(j > 100)
                    cout << "hello" << endl;
}
```

7 Close Braces Should Be On Their Own Line

Any block of code contained within a set of braces, should end with a single line with the final closing brace. This includes not only for example, for loops, while loops, and conditional block, but the end of functions as well.

```
  if(body_temp == 98.6) {
    status = "normal";
    patient_id = 0;}        // Avoid
  if(body_temp == 98.6) {
    status = "normal";
    patient_id = 0;
  }                         // Better
```

The exception is in-line function definitions in class header files:

```
class AVD_Entry
{
 public:
  AVD_Entry();
  ~AVD_Entry() {};

  bool   setSpeed(double v)    {m_speed = v;};          // This is OK
  double getSpeed() const      {return(m_speed);};      // This is OK

 private:
  double m_speed;
};
```

8 Open Braces - When They Should Be On Their Own Line

Function and class definitions always involve an open and close brace. This should always go on its own line. An example of the class definition is given above. An example for a function

```
double addTen(a) {    // Avoid
  return(a+10);
}
```

```
double addTen(a)      // Better
{
  return(a+10);
}
```

For all other blocks of code, e.g., for-loops, while-loops, and if-then clauses, the open brace should appear at the end of the first line. For example:

```
  while(!done) {
    a++;
    done = checkIfDone(a);
  }
```

```
  for(int i=0; i<100; i++) {
    if((a%2)==0) {
      cout << a << " is an even number" << endl;;
    }
  }
```

The motivation for the latter is that it simply reduces the number of nearly-empty lines containing just the brace. While having an open brace on its own line at the beginning of a function introduces a nearly-empty line, this is limited to at most one per function. On the other hand there may be many conditional and loop blocks in a single function and this can seriously stretch out the code. And this in turn means more scrolling to see the same lines of code.

9 Commenting Class Functions

Commenting functions is super subjective, but doing it in *some* way is almost universally accepted as a good practice. Here's how it has been done on over 99% of the software in the MOOS-IvP tree. More so on the IvP part of the code.

Minimally there is a single comment line delimiter and the name of the function in the following format:

```
//---------------------------------------------------------------------
// Procedure: isValid()

bool MyClass::isValid()
{
  ...
}
```

Often additional descriptions on the function purpose or return values are described:

```
//---------------------------------------------------------------------
// Procedure: isValid()
//   Purpose: Check if all three member variables have been set
//   Returns: false if any of the three are not set, true otherwise.

bool MyClass::isValid()
{
  ...
}
```

For functions that take as input a string that will be parsed by the function, we highly recommend putting an example of that string in the comment:

```
//---------------------------------------------------------------------
// Procedure: parseNodeUpdate()
//   Example: "node_name=alpha,utc_time=17002032344.3,battery_level=78.3"

bool MyClass::parseNodeUpdate(string update)
{
  ...
}
```

10 Class Definition Organization

The class definition is contained typically in a dedicated `.h` or "header" file. C++ provides flexibility on the structure or order of the class definition and we provide here our convention for laying out the class. Again, most of this is for readability and consistency across code, but there are a few solid programming practices addressed in the below example too. Here is our example class:

```
 1  class MyClass()
 2  {
 3   public: // Constructor and Destructors
 4     MyClass();
 5     ~MyClass();
 6
 7   public: // Public functions
 8     void setAlpha(int);
 9     void setBravo(int);
10
11     int  getAlpha() const {return(m_alpha);};
12     int  getBravo() const {return(m_alpha);};
13
14   protected: // Protected utility functions
15     bool isAlphaInRange(int);
16     bool isBravoInRange(int);
17
18   protected: // Member variables
19     int  m_alpha;
20     int  m_bravo;
21  };
```

1. Functions First, Followed by Member Variables
The first convention on display in this example is that all member functions are declared first (lines 3-16), followed by member variable declarations in lines 18-20.

2. Constructor and Destructors Declared First
The constructor(s) and destructors should be the first functions declared, as in this example in lines 4 and 5.

3. Public Functions First, Followed by Non-Public Functions
The public functions effectively define the class interface, i.e., these are the only functions that users of this class may invoke. For this reason we generally group them separately and first before any non-public functions are declared. If you have utility functions that are intended to be invoked only from within other class functions, by all means declare them to be protected or private. It's hard to later change them to be non-public because it make break the code of users of this class that invoked the once-public function.

4. Group Your Functions by Category Where Possible
Class functions typically fall into certain categories and readability is improved by grouping them in the class definition. Categories may include for example constructor/destructor (lines 4-5), setters (lines 8-9), getters (lines 11-12), and utility functions (lines 15-16).

5. Group Your Member Variables by Category Where Possible

Member variables can also often be grouped into categories and improve the readability of the code. This really depends on the class, but one of my favorite ways to group member variables is into two groups: variables that represent the *configuration* of the class, usually set at the time of instantiation, and variables that represent the *state* of the class which usually evolve over the operation of the program.

11 Class Names - Short and Unique and Use Camel Hump Notation

In picking a C++ class name, a balance should be struck between being long enough to be informative and short enough not to overly clutter code where it is used. Camel-hump notation uses a capital letter for each word component in the class name. By always beginning with a capital letter, class names are easily distinguishable from member functions, which also use camel-hump notation. Some examples:

- `IvPProblem`
- `MarineViewer`
- `NodeReporter`

Informative class names are good, but shorter class names are also good. For example above, we chose `IvPProblem` and not `IntervalProgrammingProblem`. The former is not in any danger of being to generic. Bad choices, in terms of being generic, are names like `Item` or `Element`. You want to avoid any problems later with name clashes.

12 Function Return Statements Use Parentheses

A purely subjective preference is that `return` statements always use parentheses. You can find vocal advocates on web forums for both styles. I chose to use them, partly because it may have been more in vogue when I started coding, partly because *returning* seems like a operation taking an argument, much like function. And nowadays it is done primarily for consistency. There is more likely a consensus that parentheses are better when returning an expression as in:

```
  return(a*(3-b));
```

I also prefer the use of parentheses even when returning a simple value:

```
  return(0);
```

13 The main.cpp File and Structure

The `main()` function is the entry point into your C++ program and is usually found in a file called `main.cpp` somewhere in your files of code. Here are a few conventions we really try to follow:

- Keep the `main.cpp` filename. If someone has to read your code, a good way to start the experience is to be able to find the `main()` function in a predictable place. If you really must deviate, at least use a filename such as `MyAppMain.cpp`.
- Keep the block of code in `main()` very short, preferably what can be fit on one screen without scrolling.
- Error check your command line arguments, and provide informative output if a required argument is missing, or if an argument fails to pass a certain test, e.g., a variable out of range or a file not found.
- Provide support for the `--help` command line argument which tells the user (a) a brief synopsis on what the program is supposed to do, and (b) what are the command line options and defaults.
- The top level `main()` function returns an integer at the conclusion of your program. Try to return 0 upon success or normal operation, and return a non-zero number otherwise.