Lab 09 - C++ Coding Guidelines

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

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 structure 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 and even more so if you are writing code to be included in the MOOS-IvP distribution.

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

- Put Your Name On Your Code
- Each C++ Class Has a Dedicated .cpp and .h file
- Initialize Your Variables
- Use Distinct Notation for Class Member Variables
- Create Class Member Variables Only When Needed
- Use const Functions Where Appropriate
- Use unsigned int vs. int Where Appropriate
- Return Bool in Parameter Handling Functions
- Avoid Overloading Variables

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 styleguide 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 Each C++ Class Has a Dedicated .cpp and .h File

When creating a C++ class, a simple rule of thumb is to implement the class definition/interface in a dedicated `.h` file and the class implementation in a dedicated `.cpp` file. Developers sometimes bend this rule when a class is regarded as a simple/small utility class only used by another more complex class. In this case the smaller class definition and implementation is embedded in the larger classes definition (`.h`) file. Even in these cases I strongly recommend using a dedicated `.h` and `.cpp` file even for small classes. By sticking to this convention, it makes it far easier for someone to track down a class when they see it being used in the code somewhere.

If a class is used in more than one place, e.g., in more than one other class implementation, then the first class certainly ought to be defined and implemented in its own `.h` and `.cpp` files.

4 Initialize Your Variables

The worst kinds of bugs are those that happen intermittently. Even worse if code is bug-free on your computer but someone informs you that it crashes on their computer. If you find yourself in this situation, avoid the temptation to say "but it works on my machine".

Intermittent bugs are often due to uninitialized variables. When the following line of code is executed:

```
double a;
```

Sometimes (maybe most of the time?) the initial value will be 0. But often it will not, and this is where the intermittent nature of the problem is introduced.

When implementing a C++ class, always look through your list of member variables in your class definition, and make sure you are initializing those same variables in your class constructor. When I say "especially" class member variables, I don't mean they are somehow more important. I say "especially" because it should be an item on your mental checklist to match up your class member variable declarations, and ensure you have the required initializations in your constructor. Another tip: initialize your member variables in the same order they are declared if possible. This makes it easer to double-check that all member variables have been initialized.

You can safely omit initializations of certain classes that have their own initializations, as with many of the STL classes such as `string`, `vector` and so on. For example, in a class with the following three member variables:

```
...
double       m_speed;
unsigned int m_amount;
std::string  m_status;
...
```

the following may be found in the corresponding constructor:

```
...
m_speed  = 0;    // necessary
m_amount = 0;    // necessary
m_status = "";   // NOT necessary
...
```

Even if you are sure that `m_speed` will otherwise be set somehow before it is referenced, always initialize it.

5 Use Distinct Notation for Class Member Variables

Class member variables variable are essentially global within any function implemented for the C++ class. We prefix these variables with `m_`. This makes code much easier to read since we immediately know if a variable was declared locally or defined at the class level. Some examples:

- `m_verbose`
- `m_line_length`
- `m_dynamic_scale`

6 Create Class Member Variables Only When Needed

Code that is simpler is easier to understand and maintain. One way to simplify code is to not introduce class member variables unless needed. Class member variables act like global variables in that they are accessible across all class functions. They provide a form of persistence because they don't go out of scope (they hold they value) even after a function call has completed. Consider the below code snippet for a fictional class `MyClass`. The class has two member variables and a function:

```
class MyClass
{
  ...
  void calcSquarRoot();
  ...

  double m_sqrt;
  double m_current_value;
  ...
}
```

And the below function:

```
void MyClass::calcSquareRoot()
{
  m_sqrt = sqrt(m_current_value);
  cout << "The square root is: " << m_sqrt << endl;
}
```

The above function implies two things to me. First it implies that the `m_current_value` variable was set elsewhere in some other function, and second, that the `m_sqrt` variable is used elsewhere in some other function. In readable, simple code, the implications should match reality. If the variable `m_sqrt` is not actually used elsewhere, it is still legal compilable code, but it is overly complex and misleading. A better implementation would be:

```
class MyClass
{
  ...
  void calcSquarRoot();
  ...
  double m_current_value;
  ...
}

void MyClass::calcSquareRoot()
{
  double sqrt = sqrt(m_current_value);
  cout << "The square root is: " << sqrt << endl;
}
```

7 Use const Functions Where Appropriate

If a class member function doesn't result in a change in value or state of any class member variables, it really should be implemented as `const` function. For example:

```
bool tempIsPositive() const     // <-- Note the use of const
{
  if(m_temperature > 0)
    return(true);
  return(false);
}
```

is preferable to the below since the member variable, `m_temperature` is only examined and not modified. This may be obvious in this short example, but in longer functions it may be hard to discern from a glance whether the function results in a state change.

```
bool tempIsPositive()
{
  if(m_temperature > 0)
    return(true);
  return(false);
}
```

Use of const functions, where possible, ultimately makes the code much more readable by conveying to readers a key property of the function, i.e., whether or not the local state of the class instance will be changed by invoking this function.

8 Use unsigned int vs. int Where Appropriate

Unsigned integers are similar to integers, usually taking up the same amount of bits. But unsigned ints don't interpret the higher order bit as the sign, but instead assume the number is non-negative and use that one extra bit to represent the value, thus effectively doubling the range of values that can be represented with the same number of bits.

The biggest advantage is that they convey to the compiler, and to later readers or maintainers of the code, the intended use of the variable. It will help the compiler help you find mistakes. You should use unsigned ints when representing (a) a *count* of some kind of event and (b) the *size* of a container. For example, the following should not occur in your code, even though the compiler may let you get away with it:

```
  int total_days = holidays.size();
```

If `holidays` is your own data structure, and a non-zero number is just not possible, the `size()` function should be implemented to return an unsigned int. If `holidays` is an instance of the STL class `vector`, or `set` for example, these classes *do* implement the `size()` function to return an unsigned int, and the above line of code is even more egregious. By using:

```
  unsigned int total_days = holidays.size();
```

this tells the reader a lot about how `holidays` is implemented, without even knowing the class type of `holidays`. So it's a win even if only for pure code readability. But even better it will allow the compiler to work on your behalf if you try to later treat it in a way that is inconsistent with the

assumption that it is non-negative.

Use of unsigned int allows you to safely assume the number will never be less than zero which can be convenient in certain cases. For example, if a function takes as an argument an index into an array, typically the function will include a check that the index is valid:

```
  double MyClass::getTemperature(unsigned int index)
  {
    if(index >= m_series_temp.size())
      return(0);
    return(m_series_temp[index])
  }
```

In the above there is no need to check that the index negative. If instead, the argument is an int, then we would need:

```
  double MyClass::getTemperature(int index)
  {
    if((index < 0) || (index >= m_series_temp.size()))
      return(0);
    return(m_series_temp[index])
  }
```

9 Return Bool in Parameter Handling Functions

If you have a function that is handling input or configuration parameters, implement some sensible correctness checking, and return true or false.

For example:

```
bool SomeClass::setHeading(double value)
{
  if((value < 0) || (value >= 360))
    return(false);
  m_heading = value;
  return(true);
}
```

is better for a function taking a value in the range $[0, 360)$, compared to the following:

```
void SomeClass::setHeading(double value)
{
  m_heading = value;
}
```