MAAV C++ Coding Style Guide

Introduction

Why do we have this style guide here? Because coding can be very unenjoyable sometimes, and working with badly written code is even more unenjoyable. In addition, merging in work when different authors use different style is much more difficult than merging in work from authors who use the same style. Getting a complex piece of software working is inherently difficult, so we want our team members to be in their best emotional states when they are coding. Since seeing badly written software makes us angry, we would like all (coding) team members to follow this style guide when they code, so as to produce software that is easy to read, work with, maintain, and extend. This style guide is based on the following concepts:

1. **The Concept of DRY**
   Every piece of knowledge must have a **single**, unambiguous, authoritative representation within a piece of software. This is known as the Do not Repeat Yourself (DRY) principle. Minimize the amount of duplicate code in your software, or we will get mad at you!
   For a thorough discussion about the principle of DRY, refer to section 7 in *The Pragmatic Programmer* by Andrew Hunt and David Thomas.

2. **The Concept of Code Reliability**
   Many bugs in software can be prevented by coding in a simple, clear, and consistent style that follows idioms and patterns distilled by programming "gurus." This style guide summarizes many advices from various C++ gurus such as Scott Meyers and Bjarne

Table of Contents

Stroustrup, so conforming to this style guide will most likely improve your code reliability.

3. **The Concept of Not Looking Clueless**
You know, your fellow teammates are all intelligent human beings. We can find out who wrote which piece of code. If you don't want to look like an idiot in front of the whole team, you should probably follow this style guide.

We think that we just gave you some good reasons for following this style guide. What, you are not convinced? Well, if you do not follow this style guide, we will not merge in your code. End of story. **So you better follow this style guide!**

General Rules

This section covers general rules such as the formatting and commenting.

Whitespace

1. Spaces around operators such as +, -, >, <, *, /, >=, <=, =, ==:
```
int x = 4 + 5;
```
2. Spaces around keywords such as `if`, `else`, `else if`, `while`, `for`, `do`, `switch`:
```
if (whatever()) //Notice that space
    ;
```
3. No spaces around functions or operators that feel like functions such as `sizeof()`, `typeof()`, `alignof()`, and C++ style casts:
```
int size = static_cast<int>(v.size());  // where v is a std::vector
```
4. Inside parentheses, there should be no spaces by the parentheses, but there should be spacing between arguments. For example:
```
tmp = a_function(blah, arg2);       //Good
tmp = a_function( blah, arg2 );     //Bad
tmp = a_function(blah,arg2);        //Bad
```
5. Space after C style casts:
```
int size = (int) v.size();  // where v is a std::vector
```
6. Blank lines should contain no whitespace.
7. Lines should not have any trailing whitespace.

Indentation

Use tabs that are 4 spaces in the width.

You *must* follow this rule. Anyone that has gone through EECS 280 has seen the 2-character space indents used by Professors DeOrio and Fu. While this looks good on slides, it's *not fun at all* in real life.

Proper indentation for declaring C++ classes:

```
class Blah
{
public:
    void whatever();

private:
    void privateStuff();
}
```

Proper indentation for switch statements

```
switch(tmp)
{
case 10:
    // stuff
    break;
case 9:
    // stuff
    // Fall Through
case 2:
    //stuff
    break;
default:
    //stuff
    break;
}
```

Breaking Lines

Limit lines to 80 characters. This is a hard limit unless going past this significantly enhances readability or breaks a user-visible string like a log message.

Breaking a line can be formatted either of these ways:

```
int tmp = super_long_function_call(long_variable_one, long_variable_two,
    long_variable_three, long_variable_four);
```

or

```
int super_long_function_call(
    arg_one,
    arg_two,
    arg_three
);
```

Braces

There are many legitimate styles for placing braces. We allow the following two styles:

1. **Kernighan and Ritchie (K&R) style**
   This is the style used in the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.
2. **Allman style**, also know as **BSD style**
   This style is named after Eric Allman, who implemented many of the utilities for BSD Unix.

For functions:

```
// Both K&R and Allman
int func_definition(int blah)
{
    //stuff
}
```

For `if` statements, `for` statements, and `while` statements, etc.:

```
// K&R
if (blah == 2) {
    //Do stuff
    //Lots of stuff
} else if (blah == 1) {
    //Do different stuff
    //Lots of stuff
} else {
    //stuff
    //Lots of stuff
}

while (blah) {
    //stuff
    //Lots of stuff
}

// Allman
if (blah == 2)
{
    //Do stuff
    //Lots of stuff
}
else if (blah == 1)
{
    //Do different stuff
    //Lots of stuff
}
else
{
    //Stuff
    //Lots of stuff
}

while (blah)
{
    //Shtuff
    //Lots of stuff
}
```

Traditionally, for both styles, if you only have one-liners it's okay to omit the braces. However, do not do it. If you see legacy code that omit braces for one-liners, add the braces.

All other styles, such as Stroustrup style, GNU style, Linux kernel style, Whitesmiths style, Lisp style, Horstmann style, Pico style, Ratliff style, are all not allowed. Also, always follow the style that already exists in the file you are editing.

Commenting

The purpose of comments is to help people understand your code: not just other people, but also yourself. If you poorly comment your code, chances are you will have a harder time writing your code. Also, if you ask us for help with your code, and your code is poorly commented and hardly readable, we reserve the rights to get angry at you and send you away to a corner, where you will clean your code up.

However, do not overcomment your code. More specifically, do not write comments that are completely synonomous with the code like:

```
// add a to i and store value in i
i += a;
// call foo with i
foo(i);
```

If you are confused as for what you should and should not comment, Professor Kieras has wrote a very good handout on commenting. You can access it here. However, do not follow his indentation styles.

We're using Doxygen to generate pretty HTML documentation based on JavaDoc syntax. Here are some rules about this:

1. For classes, please fill out at least the `brief`, `details`, and `author` tags. There can be multiple `author` tags. Please keep them at the class level, and add yourself as an author only if you make a significant contribution.
2. For methods, please use at least the `brief`, `param`, and `return` tags.
3. If there are pre and post conditions to your method, then also you can add `pre` and `post` tags in the comment.

Commit Messages

Commit messages should be formmatted as follows (thanks to Tim Pope from tpo.pe for this template):

```
Bug #<bug number here> - Summary of bug

Detailed explanatory text, if necessary.  Wrap it to about 72 characters or
so.  In some contexts, the first line is treated as the subject of an email
```

```
and the rest of the text as the body.  The blank line separating the
summary from the body is critical (unless you omit the body entirely);
tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
* Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here
```

Scott Chacon recommends using the imperative present tense in commit messages. In other words, write things as commands. Instead of writing "I added ..." or "Adding ...", use "Add ...".

Follow this within reason. If you change something so simple it only requires a one sentence summary, just write a sentence.

File Organization

(Note: This section is adapted from Professor Kieras' handout on C++ header files.)

C++ programs normally take the form of a collection of separately compiled modules. The contents of a module consist of user defined types, global variables, and functions. The functions themselves are normally defined in a source file (a “.cpp” file). Each source file has a header file (a “.h” file traditionally, but use we use ".hpp" files) associated with it that provides the declarations needed by other modules to make use of this module. The idea is that other modules can access the functionality in module X simply by `#include`eing the `X.hpp` header file, and the linker will do the rest. The code in `X.cpp` needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X’s code into the final executable without needing to recompile it, which enables the Unix `make` utility and IDEs to work very efficiently. Usually, the `main` module does not have a corresponding header file, since it normally uses functionality in the other modules, rather than providing functionality to them.

A well organized C++ program has a good choice of modules, and properly constructed header files that make it easy to understand and access the functionality in a module. Well-designed header files reduce the need to recompile the source files for components whenever changes to other components are made. The trick is reduce the amount of “coupling” between components by minimizing the number of header files that a module’s header file itself `#include`s. On very large projects (like MAAV's navigation code), minimizing coupling can make a huge difference in build time as well as simplifying the code organization and debugging.

The following guidelines summarize how to set up your header and source files for the greatest clarity and compilation convenience.

1. Each module should correspond to a clear piece of functionality. Conceptually, a module is a group of declarations and functions can be developed and maintained separately from other modules, and perhaps even reused in entirely different projects. Don’t force together into a module things that will be used or maintained separately, and don’t separate things that will always be used and maintained together. The Standard Library modules <cmath> and <string> are good examples of clearly distinct modules.
2. Always use include guards in a header file. The most compact form of include guards uses `ifndef`. Choose a guard symbol based on the header file name. Follow the convention of making the symbol all-caps. For example `Roomba.hpp` would start with:

   ```
   #ifndef MAAV_ROOMBA_HPP     // Note the word MAAV
   #define MAAV_ROOMBA_HPP
   ```

   and end with:

   ```
   #endif // MAAV_ROOMBA_HPP
   ```
3. All of the declarations needed to use a module must appear in its header file, and this file is always used to access the module. Thus `#include`ing the header file provides all the information necessary for code using the module to compile and link correctly. Furthermore, if module A needs to use module X’s functionality, it should always `#include “X.h”`, and never contain hard-coded declarations for structs, classes, globals, or functions that appear in module X. Why? If module X is changed, but you forget to change the hard-coded declarations in module A, module A could easily fail with subtle run-time errors that won’t be detected by either the compiler or linker!!! Always referring to a module through its header file ensures that only a single set of declarations needs to be maintained.
4. Do not put any form of `using` statement at the top level in a header file. The reason? Anybody wanting to use your module has to `#include` your header file. If you have `using` statements in it, then these statements become part of their code, appearing at the point your header file was `#include`d. They are stuck with whatever namespace decision you made, and can’t override it with their own. Furthermore, the `using` statement will take effect at the point where it appears in the code that `#include`d the header, meaning that any code appearing before that might get treated differently from code appearing after that point. There might be a hodgepodge of which headers and code gets interpreted in terms of your namespace decision. A single `using namespace std;` statement in a single header file in a complex project can make a mess out of the namespace management for the whole project. So, no top level `using` statements in a header file! Narrowly-scoped `using` statements are OK but discouraged. Occasionally it is useful or necessary to have a `using` statement whose scope is within a class

declaration or function definition. Because these statements have a limited scope, they do not affect the entire compilation unit, and so present no problem.
Just to be clear, by `using` statements, we mean namespace `using` statements, not type alias statements.

5. The header file should contain only declarations, templates, and inline function definitions, and is included by the source file for the module. Put structure and class declarations, function prototypes, and global variable `extern` declarations, in the header file; put the function definitions and global variable definitions and initializations in the source file. The source file for a module must include the header file; the compiler will detect any discrepancies between the two, and thus help ensure consistency.
   Note that for templates, unless you are using explicit instantiations (which is quite rare), the compiler must have the full definitions available in order to instantiate the template, and so all templated function definitions must appear in the header file. Similarly, the compiler must have the full definitions available for ordinary (nonmember) functions that need to be `inlined`, so the definitions for `inline` functions will also appear (declared `inline`) in the header file.
6. Set up global variables for a module with an `extern` declaration in the header file, and a defining declaration in the source file. For global variables that will be known throughout the program, place an `extern` declaration in the header file, as in:
   ```
   extern int NUMBER_OF_ENTITIES;
   ```
   The other modules `#include` only the header file. The source file for the module must include this same header file, and near the beginning of the file, a defining declaration should appear - this declaration both defines and initializes the global variables, as in:
   ```
   int NUMBER_OF_ENTITIES = 0;
   ```
   Of course, some other value besides zero could be used as the initial value, and `static`/global variables are initialized to zero by default; but initializing explicitly to zero is customary.
7. Keep a module’s internal declarations out of the header file. The header file is supposed to contain the public interface for the module, and everything else is supposed to be hidden from outside view and access. Thus, if you need class or struct declarations, global variables, templates, or functions that are used only in the code in the source file, put their definitions or declarations at convenient points in the `.cpp` file and do not mention them in the `.hpp` file. One common example is special-purpose function object class declarations for use with the Standard Library algorithms. Often these have absolutely no value outside the module, and so should not be simply tossed into the header file with other class declarations. It is better to place their declarations in the `.cpp` file just ahead of the function that

uses them. Furthermore, internal declarations should be `static` so that they will be given internal linkage. (Note that constants declared as `const` variables with initialization automatically get internal linkage, even if they appear in a header file, so declaring these as `static` is redundant and should not be done. This way, other modules do not (and can not) know about these internal declarations.

8. Put declarations in the narrowest scope possible in the header file. Double-check Guideline #5 and make sure that a declaration belongs in the header file rather than the source file for a module. If it does belong in the header file, place the declaration in the private section of a class if possible, followed by the protected section, followed by the public section of a class. Do not make it top-level in the header file unless it really needs to be that way. For example, a `typedef` or type alias declaration that is only used within a class's member functions should be declared in the private section of the class. If the client code needs to use the declaration, place the it in the public section of the class. Don't place it at the top-level of the header file unless both multiple classes and the client code needs to refer to it. A similar rule applies to `enum` types, functions, and function object classes such as ordering relations. These rarely need to be declared at the top level of a header file.
9. Every header file should `#include` every other header file that it requires to compile correctly, but no more. Consider a headerfile `A.hpp` where class `A` is declared. If another class or structure type `X` is used as a member variable of `A`, then you must `#include X.hpp` in `A.hpp` so that the compiler knows how large the `X` member is. Similarly, if a class `A` inherits from class `X` which is declared in `X.hpp`, then you must `#include X.hpp` in `A.hpp`, so that the compiler knows the full contents of an `A` object. Do not `#include` header files that only the source file code needs. For example `cmath` or `algorithm` is usually needed only by the function definitions in the `.cpp` file - `#include` it in `.cpp` file, not in the `.hpp` file.
10. If an incomplete declaration of a type `X` will do, use it instead of `#include`ing its header `X.hpp`. If another `struct` or `class` type `X` appears only as a pointer or reference type in the contents of a header file, or as a parameter type or returned value in a function declaration, then you should not `#include X.hpp`, but just place an incomplete declaration of `X` (also called a "forward" declaration) near the beginning of the header file, as in:

    ```
    class X;
    ```

    See this handout for more discussion of this powerful and valuable technique. Two important points:

    - If your header file has a function definition that requires a complete declaration of `X`, you can almost always eliminate the need by moving that function definition to the `.cpp` file, so that the header file only has the

function declaration. Now you can use only an incomplete declaration in the header file.
    - The Standard library has a header of incomplete declarations that often suffices for the `iostream` library, named `iosfwd`. You should `#include <iosfwd>` whenever possible, because the `iostream` header file is extremely large (giant templates!), and unless you must have function definitions in the header file that use iostream operators, there is no reason to include `iostream`.
11. The content of a header file should compile correctly by itself. A header file should explicitly `#include` or forward declare everything it needs. Failure to observe this rule can result in very puzzling errors when other header files or `#include`s in other files are changed. Check your headers by compiling (by itself) a `test.cpp` that contains nothing more than

    ```
    #include “A.hpp”
    ```

    It should not produce any compilation errors. If it does, then something has been left out - something else needs to be included or forward declared. Test all the headers in a project by starting at the bottom of the include hierarchy and work your way to the top. This will help to find and eliminate any accidental dependencies between header files.
12. The `A.cpp` file should first `#include` its `A.hpp` file, and then any other headers required for its code. Always `#include A.hpp` first to avoid hiding anything it is missing that gets `#include`d by other header files. Then, if A's implementation code uses X, explicitly `#include X.hpp` in `A.cpp`, so that A.cpp is not dependent on `X.hpp` accidentally being `#include`d somewhere else. There is no clear consensus on whether `A.cpp` should also `#include` header files that `A.hpp` has already included. Here are two suggestions:
    - If the `X.hpp` file is a logically unavoidable requirement for the declaration in `A.h` to compile, then `#include`ing it in `A.cpp` is redundant, since it is guaranteed to be included by `A.hpp`. So it is OK to not `#include X.hpp` in `A.cpp`, and will save some compiler time (the compiler won’t have to find and open the `.hpp` file twice).
    - Always `#include`ing `X.h` in `A.cpp` is a way of making it clear to the reader that we are using X, and helps make sure that X’s declarations are available even if the contents of `A.h` changes due to the design changes. For example, maybe we had a Thing member of a `class` at first, then changed it to a `Thing*`, but still used members of Things in the implementation code. The `#include` of `Thing.hpp` saves us a compile failure. So it is OK to redundantly `#include X.hpp` in `A.cpp`. Of course, if X becomes completely unnecessary, all of the `#include`s of `X.h` should be removed.

13. Explicitly #include the headers for all Standard Library facilities used in a .cpp file. The C++ Standard does not say which Standard Library header files must be included by which other Standard Library headers, so if you leave one out, the code may compile successfully in one implementation, but then fail in another. (This is a gap in the Standard, justified weakly as giving implementers more freedom to optimize.)
14. Never #include a .cpp file for any reason! .cpp files are normally separately compiled, so if you #include a .cpp file, you have to somehow tell people not to compile this one .cpp file out of all the others. Furthermore, if they miss this hard-to-document point, they get really confused because compiling this sort of odd file typically produces a million error messages, making people think something mysterious is fundamentally wrong with your code or how they installed it. Conclusion: If it can’t be treated like a normal header or source file, don't name it like one!

Naming

1. Take names very, very seriously. They are a major way for you to communicate your design intent to the next person who will touch your code!!!
2. Use CamelCase for class names. For example, `CircleDetector`.
3. Use mixedCase for function and variable names. For example, `detectCircle`.
4. Preprocessor symbols defined with #define must be all upper case:

```
#define MAAV_MEMORY_STREAM_HPP
```

5. Files with a class should be named after the name of the class. For example, **`CircleDetector.hpp`** and `CircleDetector.cpp`.
6. Files with procedural (class-less) functions should have a name that describes what it does. For example, **`math.hpp`** and `math.cpp`.
7. Do *not* start variable or function names or #define symbols with underscores. Leading underscores are reserved for the C/C++ preprocessor, compiler, and library implementation's internal symbols and names. Break this rule, and you risk name collisions leading to confusing errors. (The actual rules on reserved leading underscore names is somewhat complex; it is simplified here because there is no good reason to take a chance by pushing the envelope.)
8. Do not use cute or humorous names, especially if they don't help communicate the purpose of the code.
9. Don't include implementation details such as variable type information in variable names. Emphasize purpose instead. If you need to change the name of a variable when you change the type of that variable, then you are violating this rule!
10. Use variable names that do not have to be documented or explained. Longer is usually better:

```
int x = 10;      //Bad
int ngb = 10;    //Bad
int numGreenBalls = 10; //Good
```

Single letter conventional variable names are OK for very local, temporary purposes:

```
for (int i = 0; i < 10; i++)
    ;   //null statement
```

However, don't use easily confused single letter names, such as `I`(upper case i), `l` (lower case l), `O` (upper case o), `0` (digit zero), etc. Never reply on the font.

11. Names for named constants should be all uppercase:

```
const double PI = 3.141592653589;
```

12. `typedef` names should end with `_t`.

Usage of Built-In Types

This section covers guidelines on the usage of built-in types, or primitives.

General Rules

- Always initialize a primitive before use.

Numeric Types

1. Avoid declaring or using `unsigned` integers; they are seriously error prone and have no compensating advantage. Just try subtracting a larger `unsigned` value from a smaller `unsigned` value. If a number should never be negative, either test it explicitly or document it as an invariant with an assertion.
   However, note that when bitwise manipulations need to be done, using `unsigned int`s for the bit patterns may be more consistent across platforms than signed `int`s.
2. Prefer `double`s over `float`s. `double`s are nearly always precise enough for serious computations; `float`s may not be. Use `float`s only when memory space needs to be saved (or if required by an API). Do not assume that two `float`s or `double`s will compare equal even if mathematically they should. Instead, your code should test for a range of values rather than strict equality.

String Constants

Declare and define string constants as `const` pointers to `const` characters initialized to string literals rather than initialized arrays of `char` or `const std::string` variables:

```
// Bad
```

```
const char message[] = "Follow this style guide!";
    // When this instruction is executed, the computer creates
    // message as an array that is just big enough to hold the
    // string literal, which is already in memory, and then copies
    // the string literal into message. This wastes both time and
    // memory space.

// Bad
const std::sting message("Follow this style guide!");
    // Requires extra storage for message's internal array, plus
    // time to allocate this internal array and copy the literal
    // into it. Run-time memory footprint is at least twice as
    // large as the string literal.

// Good
const char* const message = "Follow this style guide!";
```

enum Types

1. Prefer using `enum class`es over C-style enums because `enum class`es are much better behaved. For example, there is no implicit conversions between an `enum class` and an `int`. Also, the enumerated values inside an `enum class` are scoped, so the traditional all-upper-case names used in C-style enums are unnecessary and distracting; use names that are clear and simple instead:

   ```
   enum class Fruit {APPLE, ORANGE, PEAR}; //Bad
   enum class Fruit {Apple, Orange, Pear}; //Good
   ```

2. Use an enumerated type instead of arbitrary numeric code values.
3. Use the default for how enum values are assigned by the compiler:

   ```
   // Bad
   enum class Fruit {Apple = 0, Orange, Pear};
       // Why? This is the default! Are you confused or what?

   // Bad
   enum class Fruit {Apple = 1, Orange = 2, Pear = 0};
       // What? Why? Is there something wrong with you?

   // Good
   enum class Fruit {Apple, Orange, Pear};
   ```

Code Structure

This section covers guidelines on the how to structure your code.

Variables

1. Declare variables in the narrowest scope possible, and at the point where they can given their first useful value.
2. Named constants or "Magic Numbers:"
   - Numerical or string constants that are "hard coded" or "magic numbers" that are written directly in the code are almost always a bad idea, especially if

they appear more than once.

```
// Bad
double horizontal = 1.33 * vertical;

// Good
const double ASPECT_RATIO = 1.33;
double horizontal = ASPECT_RATIO * vertical;
```

- However, if a value if set "by definition" and can't possibly be anything else, or if a value has no meaning or conventional name, then making that value a named constant can be unnecessary:

```
const int NOON = 12;    // Pointless
```

- A name or symbol for a constant that is a simple synonym for the constant's value is *stupid*:

```
const double TWO = 2.0;     // Idiotic...
```

- In C++, declaring and initializing a `const` variable is the idiom for defining a constant. Don’t use `#define` - it is an inferior approach in C++ because it does not give type information to the compiler.

3. Global Variables (note that the following restrictions do no apply to global named constants):
    - Avoid using global variables as much as you can! If you are thinking about using a global variable, think again. Then think again. Then think a third time. And if you want to used a global variable, confirm with a senior team member.
    - Do not use global variables to avoid defining function parameters.
    - Global variables are acceptable only when they substantially simplify the information handing in a program. For example, data that is shared between threads is usually global.

Macros

1. Do not use macros for anything except for include guards. You may use them for conditional compilation, but before you do so consult with a senior team member.
2. Use the `assert` macro liberally to help document invariants and help catch programming errors. Note that we waid "programming errors," not run time error. Also, note that the statement enclosed in an `assert` macro should not have any side effects.

Control Flow

1. Avoid deeply nested code.

```
// Bad
if (a)
```

```
{
    // ...
    if (b)
    {
        // ...
        if (c)
        {
            // ...
            if (d)
            {
                //...
                // Just when does this excecute?
            }
        }
    }
}

// Good
if (a)
{
    // ...
}
else if (b)
{
    // ...
}
else if (c)
{
    // ...
}
else
{
    // ...
}
```

2. Prefer using a `switch` statement to `if else if` constructions for selecting actions depending on the value of a single variable. These generally results in faster and cleaner code. However, there are several things about `switch` statements that you should be aware of:
   - `switch` statements cannot be used if `strings` or floating point values are being tested. You should always include a `default` case in a `switch` statement. This `default` case should output an error messgae or throw an exception if the control should never reach the `default`.
   - Remember to terminate each `case` with a `break;` statement. If you are arranging a fall-through deliberately, make sure you document it.

Usage and Design of Functions

This section covers guidelines for function usage and design.

When Should I Use a Function?

You should use functions freely to improve the clarity and organization of your code. (Modern computers are quite efficient with function calls, so avoiding function calls at the expense of code clarity is unnecessary.) There are two important cases that calls for the usage of functions especially:

1. **Use functions to reduce duplicate code:**
   Copy-and-pasting code means copy-and-pasting bugs, and copy-and-pasting bugs means more difficulty when debugging and modifying code. So avoid duplicate code as much as you can by defining a funciton for the duplicated code block.
2. **Define functions for significant conceptual pieces of work:**
   For example, in a spell-checking program, a function that processes a document will need to process each line in the document in sequence. So it makes sense to define a function that processes a line of the document. Similarly, the fuction that processes a line will need to check the spelling of each word in the input line. So it makes sense to define a function that check the spelling of a word.

When Should I Not Use a Function?

We just said that you should use functions freely. However, no freedom is absolute. We do not want you to define rubbish functions. What are rubbish functions? Here is a list:

1. Functions that are trivial:

```
// Stupid!!!
if (thereIsError())
    outputErrorMessage();
// ...
void outputErrorMessage()
{
    std::cerr << "There was an error\n";
}

// Good
if (thereIsError())
    std::cerr << "There was an error\n";
```

2. Functions that simply wraps around a Standard Library function:

```
// Bad
int i;
if(readInt(i))
{
    // Do something..
}

bool readInt(int& ir)
{
    std::cin >> ir;
    return !std::cin;
}

// Good
int i;
if (std::cin >> i)
{
    // Do something...
}
```

3. Swiss army knife functions:

```
// Bad!!! You can't tell what this function is doing from a call to
// this function. You might end up solving every world problem or
// killing the entire human race.
void useSwissArmyKnife(int operation)
{
    if (operation == 1)
        // act like a corkscrew
    else if (operation == 2)
        // act like a screwdriver
    else if (operation == 3)
        // act like a big knife
    else if (operation == 4)
        // act like a small knife
    else if (operation == 5)
        // act like a toe nail clipper
    else if (operation == 6)
        // make your computer pop gold
    else if (operation = 7)
        // solve every world problem
    else
        // kill the human race
}
```

Functions with const Input Arguments

There are times when you do not want a function to change the value of one of its input arguments. How this should be done depends on the type of the that input argument:

1. If the argument is of built-in type. You should simply pass it by value:

   ```
   void func(const int& i);        // Bad.
   void func(const int* const i);  // Bad.
   void func(int i);               // Good.
   ```

   The rational behind this is that, for built-in types, passing by value is usually faster than passing by reference or by pointer. And since we are passing the argument by value, using a `const` is redundant because the function cannot affect the caller's value anyway.
2. If the argument needs complex construction (e.g. `std::string`), you should pass it by reference-to-const:

   ```
   void func(std::vector<int> v);          // Bad.
   void func(const std::vector<int>& s);   // Good.
   ```

3. If the argument is a pointer, you need to think about whether you want the pointer itself to be `const` or if you want the object that the pointer points to to be `const`, or both:

   ```
   void func(const int* i); // Passing a pointer to a const int
   void func(int const* i); // Also passing a pointer to a const int
   void func(int* const i); // Passing a const pointer to a int
   void func(const int* const i); // Passing a const pointer to a const int
   ```

Other Miscellaneous Guidelines

1. Functions should be short and simple. Avoid functions that are longer than a screenful (~50 lines).
2. Inline short functions for perfomance and clarity:

```
// OK, but the compiler won't optimize this.

// in .hpp file
class Polygon
{
public:
    double getArea() const;
private:
    double area;
}

// in .cpp file
double Polygon::getArea() const { return area; }


// Better; the compiler will inline getArea() if it can
// in .hpp file
class Polygon
{
public:
    double getArea() const { return area; }
private:
    double area;
}


// Just as good; the compiler will inline getArea() if it can
// in .hpp file
class Polygon
{
public:
    double getArea() const;
private:
    double area;
}
inline double Polygon::getArea() const { return area;}
// Notice that this definition needs to be in the header file.
```

3. To tell the compiler that you are not using a function parameter in a definition, leave out its name.

```
// this will result in an annoying compiler warning about unused x
void func(int i, double x)
{
    // code doesn't use x
}

// No more annoying compiler warnings!
void func(int i, double)
{
    // code doesn't use x
}
```

This techinique should (only) be used when you are subclassing from third-party libraries such as Qt. For example, you might write a class `Yichen` that inherits from `QGraphicsItem`:

```
class Yichen : public QGraphicsItem
{
public:
    // Overide virtual function in QGraphicsItem
    void paint(QPainter*, const QStyleOptionGraphicsItem *,
        QWidget *) Q_DECL_OVERRIDE;
};
```

And if you are not using the `QStyleOptionGraphicsItem*` and the `QWidget*` argument, you should write this in the implementation (source) file to supress compiler warnings:

```
void Yichen::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *, QWidget *)
{
    // Paint Yichen!
}
```

4. For functions with different argument types that peform the same type of computation conceptually, instead of naming them differently basing on their argument types, use overloaded functions.

```
// Bad
void doCoolStuffwithInt(int i);
void doCoolStuffwithString(const std::string& s);

// Good
void doCoolStuff(int i);
void doCoolStuff(const std::string& s);
```

Usage and Design of User Defined Types

Class design is one of the most complex topics in Software Engineering and Object Oriented Design (people write books about it). There is simply no way that we can include a comprehensive guide on class designs in this style guide (I know, you are already complaining that the style guide is too long. Just imagine writing it.) So this section will only talk about some basics. If you want to read more about class usage and design, here is a list of resources that you can refer to:

1. *Effective C++* by Scott Meyers, which is in the office.
2. *More Effective C++* by Scott Meyers.
3. *Effective STL* by Scott Meyers.
4. *Effective Modern C++* by Scott Meyers, which is in the office.
5. *Code Complete*, 2nd Edition by Steve McConnell.
6. *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
7. *Head First Design Patterns* by Eric Freeman, Bert Bates, Kathy Sierra, Elisabeth Robson
8. EECS 381 handouts and lecture notes.

When you need to do class desgin while working on MAAV code, always consult a senior member, especially if you are new to class design. Do *NOT* just assume that your idea if a great idea!!!

General Principles

A class should have a limited, clear, and easliy stated set of responsibilities. If another person asks you what a class you wrote does, you should be able to answer in a few coherent sentences. If you cannot do this, your class is probably poorly designed. Here is a list of the kind classes that you should suspect:

- Classes that do things that are not closely related. For example, a class should not deal with IO and also handle Roomba detection.
- Classes that contains similar code as the code in its client. Either the class code should do the work, or the client code should, not both.
- Classes that have both getters functions and setters functions for most its member variables.
- Classes that have getter functions that returns non-`const` references to a member variable or a pointer to a member variable.
- Classes that only have one instance. Note that these can be a good idea at times. For example, Singleton classes and classes in the Model-View-Controller pattern (if you don't know what I am not talking about, look at the resources I listed above). However, a lot of times a class that only has one instance are usually bloated and tries to do everything at once. You should avoid those kind of classes.

Access Control

- Always list the members of a class in the order of `public`, `protected`, and then `private` members.
- The `public` section of a class should be the interface for the client of the class.
- The `protected` section of a class should be the interface for the derived classes of that class.
- Members that are not `public` or `protected` conceptually shoule be declared `private`.
- All member variables of a `class` should be `private`.

Usage of friends

- Limit the usage of `friends` to overloading the input (>>) and output (<<) opeartors. In all other cases, do not use `friends`.
- `friend` declarations should be considered as part of the `public` interface of the befriended `class` and should be declared in the same moule.

Usage of structs

Use `struct` instead of `class` for a simple concrete type all of whose members are conceptually `public`, and do not use the `public` or `private` keywords in the declaration. `structs` are especially appropriate if the type is going to be used the same way as a `struct` in C and can be appropriate even if the type has constructors, member functions, and operators - as long as all members are conceptually `public`.

Construction, Dopy, and Destruction

Here are some general guideline about construtors:

1. The constructor of a `class` should ensure that all member variables have meaningful initial values.
2. The constructor of a `class` should also make sure that all resources needed by an instance of a class are allocated. This is a very basic principle/technique known as Resource Allocation Is Initialization (RAII).
3. As much as possible, initialize member variables in the constructor initializer list rather than in the constructor body. Always list the members in the contructor initializer list in the order in which they are declared in the class.
4. Complex operations such as memory allocation should be placed in the constructor body.
5. Always define a default constructor when it is meaningfull to do so.
6. If you have a single-argument constructor, prefer to define the default constructor using a default parameter value in the same constructor.
7. Declare single-argument constructors as `explicit` to avoid implicit conversion from the argument type.
8. If the compiler-generated constuctor correctly initialized an object, use the compiler-generated constructor. Don't write code that the compiler already provides.
9. If construction fails, use an exception to inform the client code.

Here are some guidelines about "the Big Five:" destructor, copy constructor, copy assignment operator, move constructor, and move assignment operator.

1. Always follow "the Rule of Five." Either explicitly define all five of the Big Five, or define none of them and let the compiler supply them.
2. If compiler-generated Big Five do their jobs correctly, then use them. Again, do not write code that thee compiler already provide for you.
3. The destructor of a class should ensure that all resources used by an instance of class is deallocated. This is a counter part to the RAII technique. Some refer to this as Resource Deallocation is Destruction (RDID).
4. If copy and move operations are not meaningful for a class, declare them as

=delete to prevent the compiler from supplying them.
5. Follow the swap logic when implementing move constructor and move assignment if possible. See the example at the end of this section for an example of the swap logic.

Other Miscellaneous Guidelines

- Scope typedefs, type aliases, or enumeration types used in a class within the class declaration rather than declare at the top level of a header file.
- Keep "helper" class or structs out of the header file if possible. If not possible, try declaring the helper inside the class. If even that is not possible, you should probably fix your design.
- For class-wide constants, either use static member variables or use internally linked constants.
- When choosing overloaded operators for a class, only overload those operators whose conventional (built-in) meanings are conceptually similar to the operations to be done. Prefer named functions otherwise.
- Declare member functions const if they do not modify the state of the object.
- Using const_cast to avoid duplication between the const and non-const version of a member function is allowed. Note that this is the *only* situation where const_cast is acceptale!!!

Example

Here is an example of a class which illustrates some of the guidelines included in thie section:

```
// This class holds an integer array of any size,
// detects illegal subscripts and throws an exception,
// and implements the rule of 5 by providing copy construction,
// copy assignment, move construction, and move assignment.
// Operator+ is overloaded to sum two Array objects of the same size.

#include <iostream>
using namespace std;

class Array_Exception
{
public:
    Array_Exception (int v, const char * msg) :
        value (v), msg_ptr(msg) { }

    int value;
    const char * msg_ptr;
};

class Array
{
public:
    // default constructor creates an empty smart array.
    Array() : size(0), ptr(nullptr) { }
    // one-argument constructor - argument is desired size of the array
    // constructor allocates memory for the array
    Array(int size_) : size(size_)
    {
        if (size <= 0)
        {
            throw Array_Exception(size, "size must be greater than 0");
        }
        ptr = new int[size];
    }

    // copy constructor - initialize this object from another one
    Array(const Array& original) : size(original.size), ptr(new int[size])
    {
        for (int i = 0; i < size; ++i)
        {
            ptr[i] = original.ptr[i];
        }
    }

    // move constructor - take the guts from the original and leave it in
    // a safely destructable state
    Array(Array&& original) : size(0), ptr(nullptr)
    {
        swap(original);
    }

    // copy the data from rhs into lhs object using copy-swap
    Array& operator=(const Array& rhs)
    {
        Array temp(rhs);
        swap(temp);
        return *this;
    }

    // move assignment just swaps rhs with this.
    Array& operator=(Array&& rhs)
    {
        swap(rhs);
        return *this;
    }

    // swap the member variable values of this object with the other
    // could use std::swap
    void swap(Array& other)
    {
        int t_size = size;
        size = other.size;
        other.size = t_size;
        int* t_ptr = ptr;
        ptr = other.ptr;
        other.ptr = t_ptr;
    }


    // destructor - deallocate resources
    ~Array()
    {
        delete[] ptr;
    }

    int get_size() const
    {
        return size;
    }

    // overloaded plus operator returns an Array containing the sum of
    // the two
    Array operator+(const Array& rhs)
    {
        // must have the same size
        if(size != rhs.get_size())
        {
            throw Array_Exception(size, "LHS and RHS must have the same size
       for +");
        }
        Array result(size);
        for(int i = 0; i < size; ++i)
        {
            result[i] = ptr[i]+rhs[i];
        }
        return result;
    }

    // overload the subscripting operator for this class - const version
    // this will throw an error if the index is out of range
    // (including size == 0 case)
    const int& operator[] (int index) const
    {
        if ( index < 0 || index > size - 1)
        {
            // throw a bad-subscript exception
            throw Array_Exception(index, "Index out of range");
        }
        return ptr[index];
    }

    // overload the subscripting operator for this class - nonconst version
    // this will throw an error if the index is out of range
    // (including size == 0 case)
    int& operator[] (int index)
    {
        // This is the only place where a const_cast is allowed
        return const_cast<int&>(static_cast<const Array&>(*this)[index]);
    }

private:

    int size;
    int* ptr;
};
```

C++ Idioms

1. In general, prefer C++ facilities over C facilties. For example, use `cin` and `cout` instead of `scanf` and `printf`; use `new` and `delete` instead of `malloc` and `free`.
2. Use `nullptr` instead of `NULL` or `0`.
3. Use `bool` instead of zero/non-zero when you need a true/false variable.
4. Do not use the `struct` keyword except in the declaration of the `struct` type itself. C++ is not C. You do not have to repeat the `struct` keyword everywhere you are referring to a `struct` type.
5. Use `typename` instead of `class` in template parameter declarations. A template parameter does not have to be a `class` type, so `typename` is clearer:

```
template<class T> void f(int i, T t);       // Bad.
template<typename T> void f(int i, T t);    // Good.
```

6. Do not use the `exit` function in C++. Unlike in C, `exit` in C++ is not equivalent in effect to a `return` from `main`: calling `exit` does not ensure that all relevant destructors are called. Therefore you should only reserve `exit`s for emergency exits where leaving a mess is the only alternative.
7. Take advantage of the definition of non-zero as `true` and zero as `false` when testing pointers. Note that a value of `nullptr` will test as if it was zero or false.

```
// Clumsy
if(ptr != 0) { }
if(ptr == 0) { }

// Better
if(ptr) { }
if(!ptr) { }
```

8. Casts:
   - Casts usually imply that the design is bad and using casts undermines type safety. Try to correct the design if possible.
   - Never use C style cases; always use the appropriate C++ style cast that expresses the intent of the cast. The only exception to this rule is routine numeric conversions. Note that such numeric conversions are usually required to supress compiler warnings while interfacing with Standad Library facilities:

   ```
   // OK
   std::vector<int> v{0, 1, 2, 3};
   int i = static_cast<int>(v.size());

   // Also OK
   double var;
   int i = (int) var;
   ```

9. Prefer type aliases (with `using`) over `typedef`s because a type alias can be a template, while a `typedef` cannot. However, note that a type alias or a `typedef` should hide the implementation details of the type. A good example if the Standard Library `size_t`, which is a `typedef`. The implementation of `size_t` is platform dependent, but we don't care. We just know that a `size_t` always has the same size as that of a pointer. If your type alias or `typedef`. is not providing this kind of implementation-detail-hiding, you probably shouldn't use it.

Exception Handling

1. Exceptions should be used to handle run-time errors only. Run-time errors refer to errors caused by events outside the programmer's control. Examples include when the user enters an invalide command, when the system runs out of memory, or when network connection disapper. Use `assert`s to deal with errors due to a programmer's mistake in logic or coding.
2. Be aware of the C++ philosophy: For faster run speed, the language and Standard

components do not do any checks for run-time errors; instead, your code is expected to ensure that an operation is valid before performing it: either by selective checks or careful code design.
3. Explicitly design what a program will do in the case of errors. Avoid designs in which the program simply "muddles through" and attempts to keep going in the presence of run-time errors. It is almost always better to take positive action to either inform the user and/or stop processing and restore to a known state before resuming.
4. Use exceptions to allow a clear and simple separation of error and non-error flow of control, and to clearly assign responsibility for error detection and error handling.
5. Do not use exceptions for a "normal" flow of control. Exception implementations are too inefficient for that purpose; reserve them for true error situations where the program processing has to be stopped, terminated, or redirected in some way.
6. Local try-catches around a single function call are probably poor design compared to normal flow of control techniques. Reserve a local try-catch structure for the situation where you have to do some cleanup in this function before rethrowing the exception to the higher-level handler.
7. Use different exception types to signal different error situations rather than packing different values into the exception objects and testing their contents in the catch.

const Correctness

1. Use `const` everywhere that is meaningful and correct. See Item 3 of *Effective C++* by Scott Meyers (which is in the office) for a complete discussion about the rational of this.
2. If something turns out to be non-`const` when it is supposed to be `const`, try to correct the design rather than patch the error using `const_cast`
3. Don't declare thing that change as `const`!!!
4. Do not use `mutable` to fake constness of a member function.

Usage of the Standard Library

General Principles

You should know what the Standard Library facilities do, and should prefer them over your own DIY functions and classes. Why? Well, first of all, why waste time writing and debugging your own code when a solution is ready for you to use? Secondly, the Standard Library facilities are usually well optimized for the platform. Hence despite how smart you are, your code will most likely be slower than the Standard Library. So don't recode the wheel!

Also, trust that the Standard Library facilities will do their job correctly. More specifically, do not write code that only makes sense if the Standard Library is defective:

```
// Bad
std::string s = ""; // let's make sure the string is empty!
cin >> s;
// check and return an error code just in case this failed somehow
if(cin.fail())
{
    return 1;
}
// check that we read some characters
if(s.size() <= 0)
{
    return 1;
}
// looks like we can use the contents of s now

// Good
std::string s; // automatically initialized to the empty string
cin >> s; // will always succeed unless something is grossly wrong
// s is good to go
```

Using std::string

1. When appending a string a to another string s, prefer `s += a;` over `s = s + a;`. The rational behind this is that the second statement will create a temporary string to hold `s + a` and will then copy this value into s. On the other hand, the first statement will merely expand s and copy the value of a into the expanded memory. Thus the first statement is more efficient than the second one.
2. Avoid using `std::string::compare` when the regular comparison operators work:

```
// Bad
if (!str.compare("Hello") { } // Difficult to understand

// Good
if (str == "Hello") { } // Clear and obvious
```

Using Containers (and Arrays)

1. Use the `empty()` member function to check a container for being empty rather than comparing the `size()` to zero:

```
// Bad
if (container.size() == 0) { }

// Good
if (container.empty()) { }
```

2. Write `for` statements for containers (and arrays) in their conventional form if possible:

```
// Good - the conventional, most common form
for(i = 0; i < n; i++)      // correct for almost all cases

// Bad
```

```
for(i = 1; i <= n; i++)     // confusing - what is this about?
for(i = n; i > 0; i--)      // better be a good reason for this!
for(i = -1; i <= n; i++)    // totally confusing!
```

3. In for loops with iterators, use the pre-increment operator, rather than the post-increment operator. The reason for this is that the post-increment operator must copy and hold a value for returning; if unused, the compiler may not be able to optimize it away. The pre-increment operator does not have this problem and is usually more efficient than then post-increment operator.

```
// Bad
for(list<Thing>::iterator it = things.begin(); it != things.end(); it++)
// Good
for(list<Thing>::iterator it = things.begin(); it != things.end(); ++it)
```

4. When iterating through a container (or array), if possible arrange the iteration code to handle the empty case automatically instead of as a separate special case. The iterator interface is designed to do this already, so don't write redundant code like this:

```
if(container.empty())
    return;
for(auto iter = container.begin(); iter != container.end(); ++iter)
{
    //do stuff
}
```

Instead, do this:

```
for(auto iter = container.begin(); iter != container.end(); ++iter)
{
    // do stuff
}
```

Using the STL

Often when using STL algorithms, you will need to "pass a function" to the algorithm. There are size different ways to do this: use an ordinary function and pass a pointer to that function, use `std::bind`, use `mem_fn`, using a lambda expression, define a custom function object, or use `std::function<>`. Here are some guidelines about how to choose between them:

1. If the relevant function is already defined, then use the following:
   - If it is an ordinary function with no extra parameters, simply use it directly.
   - If it is a member function with no parameters, simply wrap it with `mem_fn`.
   - If it is either an ordinary function with extra parameters, or a member function with extra parameters, use bind to specify the additional parameters.
2. If the code is not already in a defined function, then do the following:
   - If the code is short and simple and needed in only one place, define it in-place with a lambda expression in the algorithm call and use whitespace and

indentation to make sure it is easy to read. If the lambda expression is too long or complex to be readable when written in place, then do not use a lambda expression.
  - If the code is complex, or it needs to be used in more than one place, either define a function or custom function object class to contain it.
  - If you need to store state during the algorithm execution, use a custom function object class that has member variables, not clumsy and limited ordinary functions with `static` or global variables.
3. Use `std::function<>` only if you really need to store callable objects of variable or unknown type and have good reasons not to use a simpler, lighter-weight mechanism.

Usage of Dynamic Memory

1. Avoid using dynamic memory when a varible on the stack would work just as well.
2. When using dynamic memory, all allocated memory must be deallocated by the program before terminating. In other words no memory leak is allowed.
3. Remember to used `delete[]` if you are deleting a dynamically allocated array.
4. Note that in Standard C++, the new operator will throw a `std::bad_alloc` exception if it fails. Hence you do not have to check whether the returned pointer is valid or not.

Remarks

This style guide borrowed heavily from the EECS 381 C++ Coding Style Guide.

Generated on Mon Aug 21 2017 19:16:26 for MAAV Navigation Software by doxygen 1.8.11