GOFIRST Source Code Conventions and Style Guide

Max Veit
GOFIRST Senior Software Engineer

Last Updated: February 27, 2014

0. Contents

1. Introduction

1.1 Purpose and Audience

The purpose of this document is to establish a uniform style of writing source code for all programs written for GOFIRST projects, mainly the IGVC competition. This

is intended to facilitate collaboration between programmers as well as maintainability of software, since a uniform style removes one of the main barriers to understanding the source code written by others (or yourself!).

This document is aimed at anybody interested in writing programs that will become a part of or aid GOFIRST projects. It assumes only a basic, general knowledge of programming, including the existence of different programming languages, the purpose of a source code file, and the existence of a language-specific syntax for the text contained therein. However, the style guides in Section B do assume a basic familiarity with the specific syntax of the language being described. To better understand these guides it might be helpful to find a simple introductory tutorial in the language and its syntactic constructs (such as function definitions, "if," and "for" constructs).

1.2 Typographic Conventions

Source code, variable names, command lines, and anything else that is only correct exactly as it appears (including spaces) will be shown in a `monospaced font`.

Definitions, special names, or other words to remember are set in a **bold** font (in text, as are section headings separate from the body text).

Commands you type at the operating system command line interface (ordinary terminal in Unix or GNU/Linux, Git Bash or Cygwin prompt in Windows) will be prefixed with a `$` symbol. You should not actually type the `$` or the space that follows it. It is only shown in this document to remind you that the text after it is a command (you should also see something like it in the actual terminal; the symbol is called the **prompt**). When you actually type the command, enter everything from the first word after the `$` onwards (including spaces, although not the one before the prompt).

If there is something in one of these command lines or program snippets you need to fill in yourself it will be indicated with *italics*. For example,

```
$ git config --global user.name "Your Name"
```

means you should replace *Your Name* with your actual name (but leave the quotes as they are; those aren't emphasized).

Command lines that are too long to fit on this page are continued on the immediately following line and without a leading `$`, like so:

```
    $ foo bar baz --quux --blah File Name Here
--hippopotomonstrosesquipedaliophobia
```

To type such a command line into the terminal, ignore the line break (replace it with a single space) or use your terminal's line-continuation character if you know what that is.

Longer examples of program code are typeset in a frame approximately 80 charaters wide, to mimic the actual appearance of code in a text editor. Omissions are indicated by an ellipsis (`...`) on its own line. For example:

```
VisionPacket VisionInterface:: getPacket() {
    ...
    // Exclude the sky
    rectangle(lines, Point(0, 0), Point(obsts.cols, params.hrzHeight),
              Scalar(0), CV_FILLED);

    // Find the contours and send the packet on its way
    vector<Mat> contours;
    VisionAlg:: findLineContours(lines, contours, lineOpenKrn);
    ...
    //// This comment was cleverly devised to be exactly eighty characters long.
```

The frame is the same as that used for section headings; however, the different font styles in each type of frame should be sufficient to distinguish the two types of constructs.

This last note is not really a typographical convention, but a note from the author: I sometimes slip into the first person while writing documentation or programming tips. You can safely assume that any use of the first person singular refers to Max Veit, the original author of this document. The first-person plural refers to the GOFIRST programming team as a whole.

1.3 Improvements and Changes

If you can suggest any changes to this document that you think would be helpful, I would appreciate it if you could contact me (the author) at the email address `max d veit at-sign gmail dot com` (delete the first two spaces and make the obvious textual substitutions).

2. Language Classes

This section lists the classes of languages that will be discussed in this document.

C-syntax (C, C++, Java and C#)
: This is a family of languages with differing programming paradigms. The one thing they have in common is the basic syntax derived from C, so they will generally be discussed as a group in this document. Most GOFIRST code so far uses the C++ and C# languages in this family.

Python
: This language blurs the boundaries between programming and scripting; its rapid development cycle and interpreted nature make it ideal for high-level control of large, composite programs, an area known as the "glue layer." Its syntax is much simpler and more consistent than that of the C-syntax languages. We are considering using this language in the 2014 competition code for the glue layer and more algorithmically focused sections of the program.

LabVIEW
: The language designed for use with National Instruments (NI) hardware. In the past, GOFIRST has used NI controllers and boards, necessitating the use of this langauge. NI's controllers and language are no longer used in the IGVC competition robots, so this document does not discuss the language in detail. What distinguishes LabVIEW from most other languages is that it is a graphical language; programs are created not by writing statements to a source code file but rather by visually connecting virtual blocks with wires. This graphical nature makes it difficult to set specific style guidelines for LabVIEW code. Some conventions are already imposed upon the programmer, but the visual layout of the blocks and wires themselves is completely up to the programmer. "Spaghetti code" that looks like a literal pile of spaghetti can quickly ensue if you are not careful to organize the blocks and wires in a logical, readable fashion. A general rule: If you cannot understand the code that you wrote yesterday, you need to put more effort into organizing your code layout. Of course, this rule applies to text-based languages as well!

3. Source Code Style and Conventions

This section focuses on the detailed mechanics of the style of GOFIRST source code. The term **style** as it is used here means the positioning, layout, whitespace, and naming aspects of source code not directly specified by the program's logical structure. Source code style is meant to facilitate the reading, understanding, and modification of a source code file. A source file can compile to a completely correct, consistent program and still be difficult or impossible to read. This is why programmers generally hold themselves to some sort of style guide, whether explicity written or not.

So as not to reinvent the wheel, this document references some already established and well-known standards; they are listed in Appendix B at the end of this document.

3.1 General

The general GOFIRST code style bases on two style manuals: The Java style guide for C-syntax languages [1] and the Python style guide for Python [2]; both are listed in Appendix B.

For Python, the entire style guide applies. Please read it in its entirety if you plan on working with Python at all.

For C-syntax languages, please read the Java style guide, sections 4-9. Also, observe section 3 insofar as it applies to your language, and take section 10 with a grain of salt. It contains good advice, but not all of it is applicable to or compatible with languages besides Java. Finally, I have one important disagreement with Section 4.2 of the guide: Lines should be broken *after* binary operators, not before. This is the most important conflict between GOFIRST style and the Java style guide, so it's worth mentioning it here even though it's covered in the points below. There's more on line breaking in general in Section 3.5 below, and more on line continuation rules specific to C++ in Section 3.2.4 below. For any finer points of line continuation not covered below, just remember: if you break in a way that is consistent throughout a single code file/module and that maintains the visual distinction between different functional structures within the code (e.g. distinguishes a function definition from the body), then you're probably good.

For both language classes, the specific recommendations in this section *override* any conflicting recommendations in the applicable style guide. To repeat, if there is any conflict between the points below and one of the two online style guides, this guide wins.

One exception to the above rule is C# code, for which the Microsoft coding conventions [3] apply. The C# language is not heavily used in GOFIRST projects as of the start of the 2014 season and as such, the extent to which the C# guide applies to GOFIRST code has not been extensively discussed. Thankfully, there are few conflicts regarding general (language-inspecific) guidelines. For now, both style guides are taken to apply to C# code, with conflicts being resolved in favor of the C# guide. This convention makes it easier to work with C# code within Microsoft's Visual Studio IDE, the only development environment currently used for C# code in GOFIRST. This document (as well as parts of the Java guide) serves to cover omissions or unspecific points in Microsoft's guide.

3.2 C/C++ Features

The languages C and C++ contain syntactic elements that are not present in Java, such as pointers and (for C++) templates. Also, the syntax for class definitions (especially when inheritance is involved) is somewhat different. This subsection sets conventions for using those features.

First of all, there may be some features that are not covered in this subsection (yet). For those features, the style set by the code examples and tutorials available at `http://cplusplus.com` should serve as an adequate guide until those features can be added here.

3.2.1 General Peculiarities and Tips

The C++ language has the peculiarity that two styles of class declaration are possible: One with the declarations in a header file and implementations in a C++ source file of the same name, and one with the entire class declaration and definition **inline** in a single C++ source file. The former is preferred, as it is impossible to `#include` C++ source files in other files.

Another C++ peculiarity is that multiple extension types are possible for source files. Use `.cpp` for C++ source code (implementation) files, `.h` for C or C++ header files. If the project mixes C and C++ code (not recommended unless it's absolutely

necessary), you may use `.hpp` for header files to indicate that they cannot be parsed by a plain C compiler.

The possibility of multiple inclusion of C++ header files in even moderately-sized projects can often cause problems with multiple definitions. To avoid this, use **header guards** to prevent the file from being included more than once. They look like this:

```
// Beginning of file
// Include statements, ‘‘using’’ namespace-related statements
...
#ifndef HEADERFILE_H_
#define HEADERFILE_H_
// Declarations for functions, variables, and classes
...
#endif
// End of file
```

The name of the defined constant should be related to the name of the header file in the following way: Put the name of the file in all caps, replace the period with an underscore, and add a trailing underscore. So the file `BufferThreadedP.h` should have a line like `#ifndef BUFFERTHREADEDP_H_` in it.

3.2.2 Pointers

The syntax for declaring pointer variables should have the asterisk (“pointer symbol”) next to the *type*, not the variable, like so:

```
BufferThreaded* buf;
```

This makes more semantic sense than the old C convention, which would be

```
BufferThreaded *buf;
```

as you are declaring a variable `buf` of type “`BufferThreaded` pointer”.

3.2.3 Operator Overloading

C++ lets you override the standard operators (such as +, ==, or <<) to implement your own functionality. Please *only* overload operators if it makes sense considering the *semantics* of the operator. For example, it is fine to overload + to join strings or add two vectors, but don’t make it merge two address book records!

You will find specific conventions for operator overloading in most good C++ books (or even online tutorials), but here is a summary:

For unary operators (operators that only operate on a single object, as opposed to two of the same type), it is best to implement the `operator` function as a member within the class.

For symmetric binary operators (that is, operators that operate on two objects *and* return the same result regardless of the order of the operands), use a non-member function (make it a `friend` of the relevant class if necessary). This makes it easier to show that the operation actually does obey the symmetry property.

For asymmetric binary operators (for example, the >> operator), it may make sense to use either member or non-member functions, depending on the context. If you know enough about your class to implement this type of operator overloading, you probably know which style is best. Also note there are asymmetric binary operators (e.g. the assignment operators, like = or +=) that can only be implemented as member functions.

In any case, you have the option of using `const` for any of the arguments. If you are not going to modify an argument (generally a good idea unless you're doing something like add-assignment), use the `const` qualifier on the reference. Naturally, if you *are* going to modify an argument, you can't use the `const` qualifier - but if you can avoid taking a non-const argument by making the operator a member function instead, do it.

3.2.4 Constructors, Initialization Lists, and Line Continuation

Constructors in C++ can quickly become unwiedly due to features such as initialization lists (which allow convenient initialization of an object's member variables, possibly using arugments to the constructor). This is intended as a supplement to Section 3.5 dealing specifically with C++ constructors.

If your init lists push a constructor line over the 79-character limit [1], which they often will, break the line preferably right after the colon (`:`) like so:

[1] see Section 3.4

```
VisionPacket::VisionPacket(vector<Mat> lineContours, Matx33f perspTrans) :
              lineContours(lineContours), perspTrans(perspTrans) {
    linesGround.resize(lineContours.size());
    sortByLength();
    ...
}
```

Notice the brace beginning the function body still goes at the end of the last line of the function definition; this is for consistency with our general C-syntax standards. Also notice the extra indentation of the second line of the constructor used to distinguish the continuation of the definition line from the body of the function. To make it look nicer, the start of the initialization list has been aligned with the name of the function (constructor), after the scope specifier ending in `::`. This is the preferred style unless said scope specifier is impractically long or short; in that case, just indent the init list one level (4 spaces) out beyond the normal indentation level of the function body.

If the constructor is too long to fit on one line even without the initialization list, align the continuation of the argument list with the opening parenthesis that started it, then indent the initialization list (and its continuation, if necessary) as described above. To illustrate with a really long example definition:

```
VisionInterface::VisionInterface(Matx33f cameraMat, Mat distCoeffs,
                                 Matx33f perspectiveTrans,
                                 std::string camAddress) :
                cameraMat(cameraMat), distCoeffs(distCoeffs),
                perspectiveTrans(perspectiveTrans) {
    openKrn = getStructuringElement(MORPH_ELLIPSE, Size(5, 5));
    ...
}
```

The idea is to visually separate the argument list from the initialization list. Notice also that with the proper indentation of the initialization list it is easy to identify the first line of the function definition (which is often what one is looking for when scanning through a function; the arguments should be documented in the function comment).

3.3 Commenting

Comments serve several different purposes in source code. A good, thorough enumeration of the various types can be found in Steve McConnell's "Code Complete" [4]. The Python style guide [2], section "Comments" (ignore the parts specific to Python

syntax) provides a more concise overview; please read it. In short: Comments should *not* repeat the code. Comments should *never* contradict the code. Make a priority of updating the comments to reflect the code, preferably before every commit.

Comments should not attempt to explain confusing code; please use tools such as variable names and defined constants to make your intent more clear. Comments are often helpful when used as headers for distinct logical sections of code. **Summary comments** and **documentation comments** (block comments, function comments) are more useful than **inline comments** (or one-line non-summary comments).

Every function and *every* class needs a **documentation comment** explaining what it does and how to use it, including a short description of each of the parameters and of the return type (if applicable). Such comments should be in a format that Doxygen [5] can parse. Please use the Javadoc-style convention of comments starting with `/**`. Commands start with an at-sign (@). An example of a well-formed Doxygen comment in C++ is:

```
/**
 * Find contours in the input binary image; first applying a morphological
 * open operator to remove small artifacts resulting from the detection
 * procedure.
 *
 * @param img Input binary image of detected lines, possibly including thin
 *        borders and other artifacts. This array will be modified by the
 *        morphological transformation.
 * @param contours Place to store the resulting contours
 * @param openKrn Morphological kernel for removing borders and artifacts
 *
 * @return The number of contours found.
 */
int VisionAlg:: findLineContours(
        InputOutputArray lines,
        OutputArrayOfArrays contours, InputArray openKrn) {
    // Use morphology to get rid of the small borders
    morphologyEx(lines, lines, MORPH_OPEN, openKrn);
    ...
```

The type and purpose of the returned data does not need to be explained for constructors, or if the function does not return anything (or returns `void`).

The Doxygen commenting convention for Python has not yet been decided. Two conventions are available, one is compatible with the Python `docstring` utility but does not support any Doxygen special commands. The other abandons the `docstring` convention to add special-command support.

Inline comments in Doxygen style should only be used in one of the following situations:

- After variable declarations that really, desperately, need a longer description in the documentation.
- Directly after members of enumerations ("enums") that require a longer description in the documentation.

In any case, first try to obviate the need for an inline comment by choosing a better name for the variable or member. If this is not practical, you may use an inline comment in Doxygen format, like so:

```
OutputArrayOfArrays contours; /**< Place to store the resulting contours */
```

If your comment pushes the line over the 79-character limit, place the comment on the line above (the < mark is no longer needed):

```
/** Morphological kernel for removing borders and artifacts */
InputArray openKrn;
```

For more details, see the Doxygen manual [5] section "Special comment blocks".

3.4 Line Length

The *maximum* number of characters allowed in one line of source code is 79. Don't go over it, even by one character. Doing so will annoy every other person who tries to read the code with an 80-character terminal, as the resulting line wraps visually disrupt and distort the text. Every decent language has a feature for long-line continuation; please use it.

The only exception here is the source code for documentation to be typeset with TEX/ LATEX(like this document). Since such files contain long blocks of prose, it can often be desirable to let the editor handle the line wrapping.

3.5 Line Continuation

First, if there is a logical way to get around having long lines (for example, assigning the result of a long expression to an intermediate variable with a shorter name) then do it. Do not, however, just shorten variable names to keep everything on one line. *Definitely* do not "rename" variables by assigning them to other variables with

shorter names! And finally, this should go without saying: One statement[2] per line. Do not try to stuff multiple statements onto a single line, *especially* where doing so would push the line over the length limit.

If you are stuck with a long line, you will have to break it somewhere. The Python style guide [2], section "Code lay-out" subsection "Indentation" suggests two conventions for continuing long function definitions or mult-line constructs, both of which can be extended in a straightforward way to other languages. The main concept is the alignment of the wrapped part of a list, which here means any number of terms separated either by commas or operators. One example could be the argument list in a function definition or call. Another example is a long compound algebraic or boolean expression.

Two styles can be used for breaking lists. In either case, the intent is to distinguish the continuation from the surrounding text. This can get particularly hairy in function definitions, where two levels of indentation mix. An example of the aligned ("pretty-printed") style is shown below in C++:

```
// Function declaration
Mat VisionAlg::findHueRange(InputArray img, double minHue, double maxHue,
                            InputArray openKrn, InputArray closeKrn) {
    // Start of function body
    Mat hsvImg(img.size(), CV_32FC3);
    ...
}
```

The first line following a function declaration is normally indented in one level relative to the line above it; the extra indentation is necessary to distinguish the continuation of the declaration line from the first line of the function body that would normally occupy that place. The alignment helps keep things organized and visually appealing.

An example of the same function definition with the hanging-indent convention:

[2] A statement is a logical division of code - in C-syntax languages, statements are separated by semicolons, thus multiple statements *could* appear on a single line of a source file. A similar construct is *possible* (not recommended!) in Python.

```
// Function declaration
Mat VisionAlg::findHueRange(
        InputArray img, double minHue, double maxHue,
        InputArray openKrn, InputArray closeKrn) {
    // Start of function body
    Mat hsvImg(img.size(), CV_32FC3);
    ...
}
```

Notice the empty opening parenthesis before the line continuation. This is to ensure all items of the parameter list still maintain a sort of alignment with each other. The hanging-indent style is particularly useful with a long stretch of unbreakable text before the line break (e.g. with very long function names - imagine the above function was called `VisionAlg::findTheRangeOfHuesWithBarrelsInside` and you can imagine why the first aligned style might not work so well).

Also, if the separators in the list happen to be operators instead of commas, please be consistent and break lines *after* binary operators such as && or =.

A special case of breakable long lines is the assignment of a value to a variable. If one were to interpret such a statement as a list, the variable would the first item, the value would be the second, and the = operator would be used to join them. In this case, however, if you have to break after the = operator then always indent the continuation by one level relative to the start of the statement, like so:

```
void awfulFunction(wayTooLongParameterName):
    reallyFreakingLongVariableName =
        unnecessarilyLongFunctionName(wayTooLongParameterName);
    // other statements
}
```

However, if you can break the line at another point, that may be preferable, especially if more than two lines are necessary to fit the entire statement. For example, this:

```
void awfulFunction(wayTooLongParameterName):
    reallyFreakingLongVariableName = unnecessarilyLongFunctionName(
        wayTooLongParameterName * 2,
        wayTooLongParameterName * 4);
    // other statements
}
```

looks much better than this:

```
void awfulFunction(wayTooLongParameterName):
    reallyFreakingLongVariableName =
        unnecessarilyLongFunctionName(
            wayTooLongParameterName * 2,
            wayTooLongParameterName * 4);
    // other statements
}
```

This rule applies in general as well: If you can, avoid using multiple indent levels to continue a single statement.

In any case, make sure that the language understands you are continuing a line; use whatever escaping constructs or implicit continuations are necessary to make the compiler interpret the continued fragment as a continuation of the previous line.

3.6 Whitespace

The issue of whitespace is often so contentious that it is a good idea to set detailed and consistent rules for its use throughout source code.

Please read the Python style guide [2], section "Whitespace in Expressions and Statements". For C-syntax code, hold yourself to these rules where they are applicable to your language. For Python code, you should observe the entire style guide anyway.

Also, don't leave whitespace at the end of your lines. For example, this makes it harder to copy and paste entire lines in some text editors. It is possible to set your editor to display such spacing errors.

3.6.1 Spaces vs. Tabs

For indentation, use four spaces and no tabs. To repeat, your source files should *not* contain the Tab character. Often, stupid "holy wars" flare up about what style of indentation is preferable or even superior; setting a convention throughout the project and sticking with it is the best way to avoid needless disputes, even if it inconveniences some developers' workflows. The spaces vs. tabs issue mostly comes down to personal preference and there is not much to be gained in a technical sense by choosing one convention over the other. I chose spaces because of a minor technical reason relating to the continuation of long lines of code (although you could

probably make a convincing technical argument either way): Clearly you want to align the continuation in some logical way with the preceding line, such as a continued argument list of a function that you want to align with the first argument in the previous line. Plain tabs are often a recipe for disaster (different editors sometimes throw off the indentation wildly), and mixing spaces and tabs for indentation in one file is generally frowned upon.

So please, for the sake of consistency, set your editor to expand tabs to spaces up to the next column that is a multiple of four (most decent editors should be able to do this). Those who claim to have a real problem with this in terms of readability will have to deal with it, i.e. either get used to it or take the time to program/script a solution.

3.6.2 Blank Lines

No more than necessary, but please do use them to separate logical "paragraphs" of code (e.g. 5-10 lines inside a function that perform a distinct task). At the top level, use two blank lines in between function definitions as well as between distinct sections of code with different purposes, e.g. between the `#include` section and the top-level variable or function declarations. Within a class, use one blank line between each function definition and to separate a block of declarations from the function definitions. Do not use blank lines directly after a function declaration.

3.6.3 Miscellaneous Whitespace Notes

Avoid whitespace at the end of lines. This makes it harder to copy and paste entire lines in some editors. If your editor has a feature to make trailing spaces visible, turn it on.

3.7 File Naming

Source files for C-syntax languages should be named after the main class in the file, with exactly the same case, and the standard extension for the file type (see Section 3.2 for file extensions for C++). In the case of Java, this convention is enforced by the compiler; the name of the file must be the same as the unique public class residing in that file (there can only be one). For other languages, name the file after the logically dominant or most important class in the file; this is usually (but not always) the first class defined in the file. If your language uses header files, always

give the source/implementation file the same name as the associated header file (should one exist), modulo extension.

Python follows a different file naming convention. Files are associated with Python **modules**; a module is more or less the runtime version of a Python file, and has the same name as the associated file with the `.py` suffix stripped. The naming conventions for Python modules (files) and packages are specified in the Python style guide [2], section "Naming Conventions" subsection "Package and Module Names".

3.8 Copyright Notice and Legal Issues

All source code files should have the following notice[3] as a comment in Doxygen format at or near the top:

```
/**
 * Documentation of file, if necessary
 ...
 * @author Author Name
 * Additional @author lines for additional authors, if any
 * @copyright Copyright © Year(s) GOFIRST. All rights reserved.
 * Additional licensing info, if any
 ...
 */
```

The copyright is necessary, or at least a good idea, because it protects our intellectual property rights over the code. The use of the term "intellectual property" doesn't mean the code is to be kept proprietary; on the contrary, it mostly refers to our right to be credited as the author(s) of the code. Most free-software licenses, for example, use copyright as an enforcement tool. It prevents others from ripping off the code and claiming it as their own for the purpose of personal credit or profit. As we haven't yet decided how to license our code, a copyright with all rights reserved is the safest option. Don't worry; whatever license we choose, we will make sure your authorship of the code is recognized and credited.

Claiming authorship is a somewhat subjective issue; if you have made a significant contribution to the code in the file, you may list yourself as an author. The first author listed should be the current primary maintainer of the code in the file, if such exists. In case any conflict should arise, the definition of "significant", and thus of

[3] Technical notes: Fit the comment style to whatever language you are using; just make sure Doxygen recognizes it. The special Doxygen commands (author, copyright) are preferred, but not required. The copyright sign (©) may be replaced in code by `(c)` if necessary. As a reminder, you can always copy and paste the symbol from other code files.

issues of authorship in general, are taken to be a matter of consensus among active developers of the project ("active" meaning they are active members of GOFIRST and have directly participated in the software development of the project in question during the current academic year).

If you wish to claim *copyright* over a certain section or file of code (rather than transferring copyright to GOFIRST[4] - this issue is mostly distinct from authorship credit), please first discuss it with the other active developers of the project, including GOFIRST's Director of Engineering (or, should that position no longer exist at this time, the equivalent or successor position). The issue of transfer of copyright as it applies to GOFIRST software development must still be clarified, preferably in a meeting of the GOFIRST officers and any interested GOFIRST developers. As far as I can tell, you have the right to copyright a code file as your own that you alone authored, as long as you have not waived that right in a previous written agreement. If you claim copyright, you also have the right to license the code file however you please. However, your fellow developers will appreciate it if you would let them know if you plan to claim this right.

4. Technical Notes

The notes below don't necessarily relate to the layout and style of code; however, they are still important for compatibility and standardization purposes.

4.1 File Encoding

Please use **UTF-8** for general symbol support and inter-system compatibility. No, I don't anticipate any need to write comments in Russian, Finnish, or Japanese; nevertheless, it is the modern, extensible way to encode text in a file. Also, the encoding happens to coincide with ASCII over the entire set of 128 pure ASCII characters, so you have no excuse.

Using Unicode strings in programming storage and communications is a little trickier, as some languages' support for the feature is still lagging behind. Please use

[4] Honestly, I'm not sure if such a transfer is legally possible without an explicit written agreement, or what even counts as a written agreement under copyright law. Before we discuss licensing and/or publishing our code we must also discuss copyright law as it applies to our situation and prepare a written agreement for transfer of copyright if necessary.

Unicode for all strings intented for communication with other programs, e.g. for strings sent via the network or via message queues. For communication with external hard-programmed hardware devices (e.g. an LCD status display), please use whatever encoding the device accepts by default.

4.2 Line Endings

Line-endings usually aren't a problem if your editor is not Notepad (if it is, shame on you; go see Appendix A). However, in order to preempt any possible problems or inconsistencies, please use the automation facilities available to you. The Git version control system includes transparent handling for line ending formatting. If you are working on Windows, make sure you set Git's `core.autocrlf` setting to `true` (You can do this by typing

```
$ git config --global core.autocrlf true
```

into the Git Bash prompt).

4.3 Git setup

Please set up your Git settings so that your commits are tagged with your name and email. This is useful if we want to know who made a certain change e.g. to find out what it was intended to do. You can set this up by typing:

```
$ git config --global user.name "Your Name"
$ git config --global user.email
your.address@emaildomain.tld
```

into the Unix or Git Bash command line.

4.4 General Rules

We have programmers using different systems (primarily *nix, but occasionally Mac and Windows), so if you have a choice between convenience and compatibility you should almost always choose compatibility.

5. Miscellaneous Recommendations

This is a miscellaneous collection of recommendations concerning programming in general, and not any specific language.

5.1 Version control

Just use it. Learn a modern version control system (in GOFIRST we use Git) and use it for every software project on which you ever work. These systems give you a complete history of the revisions your software has seen (as long as you commit often enough!), allowing the recovery of lost work, faster fixing of bugs, and in general saving you hours and hours of development time in every project, big or small. A detailed discussion of how version control should be used in GOFIRST projects is outside the scope of this document.

5.2 "Magic Numbers"

It is generally considered bad practice to embed numbers other than 0, 1, or -1 directly into a program's control flow, rather than as constants at the global, file, or class level. Other numbers (integers!) that are tied closely with the structure of the surrounding code block (for instance, as an increment for a step of a specific algorithm) are exempt from this rule *if* this use is documented with a comment briefly explaining the significance of the value. Just writing a statement like `increment = sensorLevel * 67` (example in Java) will leave anyone else who reads the code scratching their head trying to figure out what on Earth the value 67 even *means* in this context. Even worse, if the value 67 is used at multiple points throughout the code and later tests determine that a value of 72 gives more accurate results, replacing every single occurrence of the value would be a pain. And no, find-and-replace is not a good alternative. If the example were using a more common value, such as 4, you really wouldn't want to replace *that* throughout your project. It would be much better to first declare 67 as a constant at the top of the class: `public static final int sensorScaling = 67`, then using `increment = sensorLevel * sensorScaling`. This way, the semantics of the sensor scaling value are made a bit more clear (and can be made even more clear by a comment next

to the constant). Also, changing the scaling to 72 is nearly trivial, and tuning this value experimentally is much easier than it would be in the previous example.

5.3 Case Sensitivity

Do not rely on letter case to distinguish between two unique identifiers. For example, if you have already defined a variable called `httpConn`, don't make another variable called `HTTPconn`. Many languages regard these two identifiers as distinct, but you shouldn't. For one, it's hard to remember exactly what case you used for what variable. It's also not portable to case-insensitive languages or situations where the case is stripped from the name (as in the case of names of constants used in header guards).

5.4 Defaults, or Explicit Is Better Than Implicit

Just because a certain code construct or statement is redundant in a specific case does not mean you should leave it out, even if the two forms are equivalent *after* compilation. In some cases, leaving out such redundant statements can greatly hinder source code readability. For a simple example, take variable initialization. Java automatically initializes fields (variables belonging to a class) to a default value of `0`, `false`, or `null`, depending on the type. Do *not* rely on other people (or yourself, some time later) to recognize this fact immediately! Another programmer reading the code may be confused by the fact that a variable is used apparently without being initialized, an error in many other languages. Worse still, if this hypothetical programmer doesn't see the code that relies on this implicit assignment, they may change the variable's value elsewhere, creating an incredibly elusive bug. In addition, programmers with relatively little Java experience may not always remember the exact initial assigned value for a field. With integers and objects this is usually easy to figure out, but the boolean value `false` is not at all intuitively obvious.
Note: Be more careful if above-mentioned redundant statements can actually worsen the performance of the code, as opposed to those that are automatically compiled away to a more efficient form. Consider the relative impact of readability vs. performance to decide the best option.

A. Editor and IDE recommendations

This section lists some text editors and Integrated Development Environments (IDEs) that members of GOFIRST have found to be useful and productive for programming diverse languages on different platforms.

A.1 Text editors by platform

A.1.1 Windows

As mentioned above, you want to move away from Notepad. It's not even good for editing configuration files. One alternative is Notepad++ (http://notepad-plus-plus.org/).

A.1.2 Mac OS X

The built-in text editor is usable, but unsuited to programming. Luckily, quite a few cross-platform editors integrate well with OSX. You could also try TextWrangler (http://www.barebones.com/products/TextWrangler/), although I have personally never used it.

A.1.3 GNU/Linux, other *nix

The built-in editors GEdit (GNOME) and Kate (KDE) are both good, extensible, full-featured, and usable for programming in many languages. However, it might be worth it to take a look at some of the cross-platform editors as well.

A.1.4 Cross-Platform

There are two classic editors for Unix-like systems that have been extensively ported:

- Vim, the "improved version" of Vi (http://www.vim.org). Max's editor of choice; many people claim it's by far the most efficient editor they've ever used. However, Vim is notorious for its learning curve. If you have a free

weekend and feel like learning to use the editor efficiently, check out the program "vimtutor" which should be installed along with the editor (try typing `:help tutor` after starting Vim). Features include everything you'd expect from a programmer's text editor (syntax highlighting, search/replace with regexes, code folding, and diff to name a few). Built-in support for most known programming languages, often with extensions available for specific languages. Originally designed for use in a terminal, although good graphical versions are available on the download page.

- Emacs (http://www.gnu.org/software/emacs/), perhaps the more full-featured (according to some, bloated) of the two classic editors. Some people (such as Max; now you know where he stands in the "editor wars") find it cumbersome to use, although if you're a boss with chained keyboard shortcuts in Word or whatever you might like this editor. Lots of programmer's features and extensions available. It even has a built-in psychologist (try `M-x doctor`) in case its keyboard shortcuts give you a nervous breakdown. Usually pre-installed on most *nix systems, available via the terminal and sometimes graphically. For other systems, see http://www.emacswiki.org/emacs/CategoryPorts and select the version that best fits your system and needs.

In addition, jEdit (http://www.jedit.org/) is a full-featured, cross-platform text editor with support for many programming languages. It appears to to work well for editing Java code.

A.2 IDEs by language

A.2.1 Java

- Eclipse (http://www.eclipse.org/) is a popular cross-platform IDE for Java and C++ development. It has all the features you'd expect from a standard IDE, plus a (somewhat clumsy) plugin installation system. It can integrate with version control systems such as Git (which we will most likely be using) through the plugin framework. It's a large piece of software – it's several hundred megabytes and slow to start up on any system. However, many Java (and C++) programmers think the rich feature set, which includes integration with a sort of task-tracking software, is worth it.
- NetBeans (http://netbeans.org/) is definitely more lightweight than Eclipse but specializes on Java. It also has a plugin system with a large database of user-contributed plugins (Git support is naturally among them). Some pro-

grammers might find the relatively minimalistic approach of NetBeans more appealing than the one-stop-shop philosophy of Eclipse.

A.2.2 C/C++

- As mentioned above, Eclipse also works well with C++.
- If you're on a Mac, you could check out XCode (it costs $ 5 on the Mac App Store). However, it's frustrating to use for development not involving a shiny Mac application.
- On Windows, Visual Studio is the standard all-in-one IDE available for CSE students for free at http://cselabs.umn.edu/software/msdnaa. Although designed primarily for Windows development, it's usable for general C/C++ program development.

A.2.3 C#

- Those with Windows should use the Microsoft Visual Studio IDE.
- Those who don't have Windows (or don't want to install it) can try the Mono development environment, see http://www.monodevelop.org/.

A.2.4 Python

First, it's perfectly possible to develop Python with nothing but a text editor and terminal (that's what I do, and I prefer it to using an IDE). However, if you're used to using IDEs and want to use one for Python, there is one available: IDLE should be installed with the default Python installation (Linux users might need to install an additional package). It's about as minimalistic as IDEs get; nevetheless, it does simplify and integrate the Python development process. Users new to Python or lacking a decent command-line interface might find this helpful.

A.2.5 Multi-Language

If you like lightweight IDEs, try Geany (http://www.geany.org/). It's not tied to a specific language, which is both an advantage and a disadvantage depending on your personal preferences. In any case, it's got pretty much everything you need to edit and build most types of source code effectively.

A.2.6 Other

Search the web and use your best judgement ☺.

B. Annotated Bibliography

These are the documents and Web resources referred to throughout the body of this document, more or less in order of appearance. Also listed here are a few lighthearted presentations of general tips on programming.

[1] Sun Code Conventons for Java: http://www.oracle.com/technetwork/java/codeconventions-150003.pdf
This is a good, but not perfect, introduction to basic style for Java source code. It provides a guide for the general style and layout of source code files in C-like languages so programmers don't have to fight with something as trivial and infuriating as brace placement. Please read this thoroughly if you have no experience with C syntax (come on, it's only 18 pages. A TL;DR response will disqualify you from our programming team). If you have programmed in the C-syntax languages before, at least skim the guide to see what specific conventions well be using.

[2] Style Guide for Python Code: http://www.python.org/dev/peps/pep-0008/
A very good guide for Python code with general advice that applies to other programming languages as well. Please at least read the introduction, first section, and section on comments. If we do any coding in Python, the style should follow this guide.

[3] Microsoft's C# Coding Conventions: http://msdn.microsoft.com/en-us/library/ff926074.aspx
A short guide that enumerates coding style guidelines, mostly those applying to constructs specific to the C# language. The relationship between the C# conventions and this document has not been formally set. Thankfully, there are almost no conflicts regarding general (language-inspecific) guidelines.

[4] Steve McConnell's "Code Complete," 2nd Ed. (Microsoft Press 2004). Amazon link here, but no guarantees on link persistence.
The respected and well-known guide to software development, covering every

detail from project conception to commenting style. Last I checked, it is still the standard textbook for the CSCI 3801W (“Program Design and Development”) course at the University of Minnesota, a required course for all computer science majors.

[5] Doxygen Manual: http://www.stack.nl/~dimitri/doxygen/manual/index.html
A thorough guide to the use of Doxygen in any type of code. Includes tutorial information such as program usage, commenting formats and feature lists, as well as an ostensibly complete reference. Covers usage specifics for C-syntax languages, Python, and a few others.

[6] Pasta Code: http://www.gnu.org/fun/jokes/pasta.code.html
If you’re doing object-oriented programming, please try for ravioli code. Canderli code and Ristto code should be avoided because of poor maintainability. Polenta code is nice, too, if it never needs to be modified (which is generally never the case, so watch out!)

[7] Good Code: http://xkcd.com/844/
Please try to stay in the "Code Well" loop. Yes, requirements do change, and we’ve had to throw out a project or two as a result, but trust me: It’s not always that way!