Coding Convention for ECON 41701

Benjamin S. Skrainka

The Harris School of Public Policy
University of Chicago

skrainka@uchicago.edu

1 Introduction

Writing software with a cogent programming style produces code which is easier to maintain, extend, debug, and understand. You actually save time by taking the time to follow a sensible coding convention. You should attempt to follow this style guide when working on your homework. Where something is unspecified, exercise your best professional judgement. For further discussion of programming style, see Kernighan & Pike. You may also find the Google coding convention helpful (`http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`).

2 Names

Construct names using CamelCase (e.g., someVariableName) and not with underscores separating words in names (e.g., some_variable_name). With CamelCase, you should capitalize the first letter of every word except the first. Choose names which describe what the name refers to.

2.1 Variable Names

If the name refers to a variable, you should choose the first letter or two as:

n the variable name is an integer count of something, like nHH for the number of households.

v the variable is a vector, e.g., vMarketShares

m the variable is a matrix, e.g., mProductChar

f the variable is a file or file stream object, e.g., fInput dw the variable contains double precision floating point data, e.g., dwPrice. Now, I am somewhat sloppy and use this for any floating point quantity, but if you want to be more precise, then use **dw** for double word (8-bytes, e.g. a double in C++), **w** for single word (4-bytes, e.g. a float in C++), **ldw** for long double (16-bytes, e.g. long double in C++).

sz for a string – either char * or a std::string. Historically, this meant a null terminated string

The larger the scope of the variable, the better the name needs to be. For example, in a short function or loop, **p** may represent price without confusion. Over larger scales, it is a horrible name. Is it price, a pointer, or ??? Err on the side of being overly clear. Remember, you may have to put your code down for a quarter because of other obligations, such as teaching, and you want to make it as easy as possible to resume work on your code at a later date.

Define or declare only one variable per line. Follow the declaration with a short comment explaining the variable. E.g.,

```
double dwObjFuncOpt ;  // Value of objective function at optimum
```

2.1.1 Loop Indexes

Many people use i, j, k, l, etc. for loop indexes. These names are hard to search on. Consequently, I like ix, jx, and kx which make it clear that they are an index and stand out in your code

2.1.2 Pointers

Variables which are pointers should begin with **p** and have one p for each level of indirection. E.g.,

```
char *       pName ;
char **      ppName ;
BLP_CONFIG * pBlpConfig ;
```

Using this convention will help you avoid errors when dereferencing pointers.

2.1.3 Constants

Traditionally, constants are given names which are all in capital letters. This is particularly true in older variants of C, where 'const' doesn't exist and #define is used to create constants:

```
#define N_HH_TYPES       (10)
```

However, it is better to use const because the compiler can perform type checking:

```
const int N_HH_TYPES =    10 ;
```

Make sure constants – whether numeric or strings such as filenames – are defined in one place only. For languages which lack constants, such as MATLAB and Stata, the best you can do is store these values in a variable.

2.1.4 User Defined Types

When defining your own types, simple types should have a lowercase name which ends in **_t** whereas classes should be in upper CamelCase with a capital first letter and a terminal **_T**, e.g.:

```
typedef unsigned long size_t ;
class BLPSnoptProblem_T : public snoptProblem
{
  .
  .
  .
} ;
```

These suffixes make it clear that the types are user-defined or part of a system library in the case of size_t.

2.2 Library Names, Function Names and Arguments

Libraries (modules) should be named with a short word or two followed by **Lib**. The short word or two should make the purpose of the library clear. The names for all functions in that library should begin with the same word(s) which begin the library's name. This makes it clear to which library a function belongs so that you can quickly find which file defines a function. After the library name which starts the function, you should choose a verb which describes what a function does. For functions which return TRUE/FALSE, the verb should be **Is,** e.g., loadIsValid to check that the data which was loaded is valid. For example, a library for computing quantities used in the BLP model of differentiated products is named blpLib. It includes functions blpCalcShares, blpCalcCondShares, and blpCalcMeanUtilJac. blpLib has both a header file blpLib.hpp and an implementation file blpLib.cpp.

A function's arguments should first list the input arguments and then the output arguments. List arguments from most general to more specific. Each argument's name should end with an underscore to make it clear that it is an argument. Try to write functions which do one thing and do it well. Such functions are easier to debug and compose with other functions to perform more complex tasks. If you function is longer than a screen or two, you probably need to break it up so that it is easier to understand.

In general, a function should return void or status information. More complex results should be returned either as a reference or a pointer (Google prefers the latter).

3 Module Layout

First, C++ supports a range of suffixes for C++ header and implementation files. In this course, we will use .hpp for header files and .cpp for implementation files. You may also wish to put your code in its own namespace. E.g., my BLP code is in namespace blp so that it will not clash with any other libraries which I might use in case I choose the same name as some other library.

3.1 File Layout

Every file must begin with a one-line comment on the first line of the file describing what the file does. If you cannot summarize what you are doing succinctly, you are not ready to start writing code. Follow the one line comment with a larger comment describing the purpose of the file, special considerations, etc. Cite papers where relevant. You may find it helpful to include a modification history which explains why you modified the file and the date. Explain why you did something not what you did. This information can also be stored in the version control system, so a modification history is less important now, but may save you time, especially if you distribute your code. Here is an example from blpLib.cpp:

```
/* blpLib.cpp - implementation of functions needed to estimate the BLP model with MPEC
/*
 * blpLib.cpp implements the functions used to estimate the
 * BLP model of differentiated products (Berry, Levinsohn, & Pakes, 1995)
 * using MPEC (Su & Judd, 2012). Because we use MPEC,
 * most of the 'action' is in the constraints.
 *
 * In particular, blpLib.cpp computes values for the objective function,
 * constraints, gradient of the objective function, Jacobian of the
 * constraints, and Hessian needed to compute standard errors.
 *
 * Compile with -DBLP_DEBUG to enable diagnostic error messages about share values
 * which are zero, Inf, or NaN.  Primarily, these messages apply to the
 * calculation of the constraints and their Jacobian.
 *
 * modification history
 * --------------------
 * 27mar2011 bss re-enabled #if BLP_DEBUG handling of Infs to
 *               improve performance of Release build.
 * 11nov2010 bss support for portable, BLP floating point type.
 * 09nov2010 bss better handling of NaNs.
 * 30oct2010 bss implementation of many blp_* functions.
 * 29oct2010 bss written.
 *
```

```
    */
```

Note how the the modification history (or 'mod hist') consists of the date, the initials of who made the change, and a description. These descriptions could be better. See how the simple comment about compiling with -DBLP_DEBUG could save you time when building the file.

3.2 Header Files

Header files should start with the same one line comment at the .cpp file and be composed of logically separated sections for includes, declarations, etc. In addition, you should use include guards to prevent including your header file excessively or in an infinite loop (will be discussed later in the course). Lay out the module as:

```
/* myLib.hpp - trenchant one line comment */

/*
 * More lengthy comments go here!
 *
 * modification history
 * --------------------
 * 25mar2012 you written.
 *
 */

#ifndef __INCyourLib
#define __INCyourLib
//----------------------------------------------------------------------
// Includes
#include ‘‘someLibYouNeed.hpp’’
#include ‘‘anotherLibYouNeed.hpp’’

//----------------------------------------------------------------------
// Constants

// define any constants you need

//----------------------------------------------------------------------
// Types

// declare any special types or classes you need here

//----------------------------------------------------------------------
// Declarations
// function declarations go here.


#endif  // #ifdef __INCyourLib
```

4 White Space, Comments, and Braces

Use white space, comments, and braces to indicate which chunks of code go together.

4.1 White Space

4.1.1 Horizontal White Space

Put white space inside of parenthesis and around operators.

4.1.2 Indentation

Use two spaces when indenting. The text recommends three which is insane. Also, setup your editor to convert tabs to whitespace because tabs are evil. To make this easier, I set the following in my ~/.vimrc:

```
:syntax on
:set ts=2
:set ai
:set nu
:set et
```

If you are using MacVim, you can put

```
:set guifont=Menlo\ Regular:h16
```

in your .vimrc to set the default font size.

4.1.3 Vertical White Space

Use one or two newlines (vertical white spaces) to show that some lines are logically related. This is like breaking a document up into sentences, paragraphs, subsections, sections, and chapters.

4.1.4 Comments

- Comment anything which is not obvious
- When you are not sure of the correct implementation or it may need to change, mark it with a comment // XXX. This makes it easy to search for.

- You may find it easier to write the comments first to work out the general layout of your code before writing the code itself!
- Use comments with 20, 40, or 80 repetitions of '=', '-', '*', etc. to show breaks or logical groupings in code.

4.2 Braces

Line up braces vertically without indentation. E.g.,

```
int myFunc( int nArg_, double dwArg_ )
{
  int nRetVal ;     // Return value
  .
  .
  .
  if( dwArg_ > 0 )
  {
    nRetVal = 1 ;
  }
  else
  {
    nRetVal = 0 ;
    for( int ix = 0 ; ix < nArg_ ; ++ix )
    {
      nRetVal += ix ;
    }
  }
  return nRetVal ;
}
```

With this convention, it is easy to see that you have not forgotten a brace. The alternative is 1TBS (the One True Brace Style) and is to be avoided:

```
int myFunc( int nArg_, double dwArg_ ) {
  int nRetVal
  .
  .
  .
  if( dwArg_ > 0 ){
    nRetVal = 1 ;
  } else {
    nRetVal = 0 ;
    for( int ix = 0 ; ix < nArg_ ; ++ix ) {
      nRetVal += ix ;
    }
  }
  return nRetVal ;
}
```

5 Miscellaneous

Some other advice:

- Use const wherever possible. It will help the compiler catch errors
- Use C++ casts, not C-style casts, e.g. **static_cast<double> (nHH)** instead of **(double) nHH**.
- Compile with as much warning information as possible, e.g. **g++ -Wall -pedantic ...**
- When applying a prefix/postfix operator in a way without side effects, prefer the prefix version of the operator because it is faster on complex objects, like iterators. E.g., ++ix is better than ix++.
- Use parenthesis around any mathematical expressions other than the +, -, *, and / so that you do not have to worry about order of operations.
- Avoid parameterized macros where possible. If you must, then make sure you put parentheses around the arguments:

  ```
  #define max( a, b ) ( (a) > (b) ? (a) : (b) )
  ```

- Because it is easy to forget an equals sign when testing for equality, specify constants first so that the compiler will generate a warning if you make this mistake:

  ```
  if( 0 == nFound )
  {
    handleError() ;
  }

  instead of

  if( nFound == 0 )
  {
    handleError() ;
  }
  ```

- In MATLAB, use cell arrays to help set off sections and facilitate rerunning subsets of your code. To create a cell array, just start the line with %%.
- R seems to work better if you use 1TBS...
- Python uses indentation to indicate blocks. It is probably best to follow the official Python style guide.

- With Stata, do the best you can and exercise good judgement.

Built March 25, 2012