C Language Coding Standard

For
CS1003, CS1013, and CS2635

Rick Wightman

Faculty of Computer Science
University of New Brunswick
Fredericton, NB

January, 2001

INTRODUCTION

Programming is a craft. Skill in a craft requires artistic talent, or creativity, and the application of discipline. The purpose of a program is to communicate a fully specified set of instructions to the computer; however, there is an equally important requirement for a program to communicate to other programmers: it's author, the development team members, the maintenance programmers over the entire lifespan of the program. Also, within education and open source communities, source code is treated as "educational text" with the expectation that it be treated with the same care as a traditional textbook. The skill of discipline is part of this process and is formally expressed through coding standards. Standardizing the mechanics of code construction allows the programmer to focus on the creative aspects knowing that others can easily appreciate and understand what is intended and how it is accomplished.

To liken this process to writing an essay: if you don't have to think about how to form thoughts into sentences and then paragraphs; if you have a command of the language and you understand its grammar and syntax, then you can focus all of your efforts on expressing yourself effectively and the essay will flow with relatively little effort. If you are missing this skill, then readers spend their time trying figure out what exactly you are trying to say instead of following your train of thought and thinking about what it means. More immediately, the programmer spends much more time programming.

So coding standards are important. What follows is a set of coding standards appropriate for use within undergraduate courses in the UNB Faculty of Computer Science curriculum that use the C programming language. They are not intended to be definitive. They are intended to provide guidance for small programming efforts typical in undergraduate courses.

The coding standard is discussed in terms of overall program source file organization, function organization, statement organization and naming of functions and variables. Lastly there is a discussion of the role of library functions within courses. Following this, an example program attempts illustrates the implementation of the coding standard.

SOURCE FILE ORGANIZATION

There can be many sections to a program source file. The following discussion presents a description of each section. The ordering presented here should be used. Not all of the section types will necessarily be used in all programs, in which case don't put them in. Each of the sections should be clearly identified within the program source file using comments and white space (blank lines).

File Documentation

The first item in a source file should be a comment block identifying the name of the file, it's author and what functionality the code provides. Each individual function within the file should also have a comment block naming the function and the functionality it provides.

Preprocessor Information

This section should list the header files that are needed (`#include`), followed by the preprocessor macros (`#define` as well as others). Preprocessor macro names should use capital letters only.

Type Definitions

These are programmer defined data types, named using the C keyword `typedef`. Within some programming teams, `struct` definitions, without the associated variables but with the fields

listed are also heavily used. This avoids introducing both a `typedef` identifier and a structure for infrequently used structures

Function Prototypes
Prototypes for all programmer defined functions should be presented. The arguments should be specified with both data type and name (e.g. `float fXValue`).

The main Function
This is the main program (function) body. It should be preceded by a comment block describing what it does. This function must be present in programs contained in a single source (`.c`) file. For larger programs divided into several source files, it must be present in exactly one source file.

Functions
Functions should be listed last. Ordering of the functions themselves is at the discretion of the programmer; however a top-down approach is suggested.. This means that the first functions defined should be those that are called from the main and that function calling no other function are listed last. Each function should be preceded by a comment block listing what duties the function performs.

FUNCTION FORMAT

As already stated, the function should be preceded by a comment block. All functions except the main will have a prototype listed in the prototype section of the program source file. The main function must be of type int (i.e. `int main()`)must return an appropriate completion code. The following discussion should be applied. The function should have at least three identifiable sections: variable declaration; function code and function return.

Variable Declaration
All variables to be used in the function will be listed immediately following the function's opening brace. Variables will be listed, one per line, with an inline comment describing its purpose. All variables within a program source file are to be declared within a function. *Global variables are **not** to be used, except in specific circumstances accompanied by careful discipline and extensive style rules that are outside of the scope of this document.*

Function Code
The function code performs that task required of the function.

Function Return
The return statement (`return`) should appear as the last statement in the function. The return statement may be optional for functions of type `void` for some programming projects.

STATEMENT (BLOCK) FORMAT

Statement (code) blocks should set off by indenting using a tab. The opening brace of a statement block should stand by itself in a line, as should the closing brace with neither brace being themselves indented. An inline comment may be useful after the closing brace to identify what it closes.

Within a statement block it may be useful to separate tasks using blank lines.

Statements that exceed the width of the screen (or the width of a printed page) should be broken into

more than one line so that the continuation line is indented from the parent line.

NAMING

This is an incredibly important area that beginning (and other) programmers overlook. The naming of a function or variable should leave little question about the purpose it serves within the program. There are several different, widely used standards for "naming", i.e. *choosing* identifiers**.** It will be necessary to follow whichever standard is prescribed for the project. Course work should follow the standard in this document unless the instructor specifies otherwise.

Functions

Function names should be meaningful. Names made up of multiple words should have the initial letter capitalized, except for the first word (e.g. `calcFutureValue`).

Variables

Variable names should follow the same pattern as function names with one additional complication: the variable name should be prefaced by data type identifiers. The identifier scheme is as follows:

Name	Data type	Prefix	Example
Character	char	c	cInitial
Short Integer	short	i	iDiceValue
Integer	int	i	iNumberOfMarks
Long Integer	long	l	lWorldPopulation
Single-precision Real	float	f	fStudentMark
Double-precision Real	double	d	dXCoordinate
String	char * or char []	s	sStudentName
Pointer	<type> *	p	plWorldPopulation

Style

The use of side-effects is discouraged since it detracts from readability and comprehension. There should be no more than one statement on each line. White space (blank lines) should be added to space out statements for added readability.

LIBRARY ROUTINES

Use of Library Routines

The student is expected to use library routines that have been covered in the course, or in a prerequisite course, when they will simplify the program. For example, rather than write code to copy a string, students would be expected to use `strcpy()`.The exception to this rule would be when writing this code is part of the assignment (e.g. "Write a function that demonstrates how the `strcpy()` function might be coded."). Generally C programmers should make heavy use of the string (e.g. `strcpy()`), character (e.g. `isalpha()`) and standard i/o (e.g. `printf()`) library

functions.

Prototypes for Library Routines

Use of the correct prototype for each library routine is required. This prototype must be supplied by providing the correct header (`.h`) file by means of an `#include` statement. Under UNIX and Linux, the command `man` command can be used to determine the correct .h file. Most compilers include an option (e.g. `-Wall` for `gcc`) that reports on failure to provide a prototype.

The `gets()` Library Function

Use of the standard i/o function `gets()` is ***absolutely prohibited*** since it introduces an unavoidable security hole and/or bug into every program it is a part of (see, e.g. Stevens, W. Richard. 1993. Advanced programming in the UNIX environment. Addison Wesley. Pp 130-131).

SAMPLE PROGRAM SOURCE FILE

```
/*
 *    template:   a template for the construction of C language
 *                program source files. Presents a solution for
 *                calculating the length of a line from its
 *                composite segments
 *
 *    Author:     Rick Wightman, 61340
 *                CS1003
 *                University of New Brunswick
 *                Fredericton, NB
 *
 *    Created:    24 December, 2000
 *
 *    Modified:
 */

/** REQUIRED HEADER FILES     */

#include <stdio.h>
#include <math.h>


/** MACRO DEFINITIONS   */

#define MAXCOORD 100


/** FUNCTION PROTOTYPES */

float calcHypotenuse(float x1, float y1, float x2, float y2);


/** MAIN PROGRAM  */

/*
 * main:    Accepts x y values from the keyboard and calculates the
 *                hypotenuses (distances) between the coordinates
 */
int main(int argc, char* argv[])
{
      float fXCoord[MAXCOORD];      /* x values from input  */
      float fYCoord[MAXCOORD];      /* y values from input  */
      int   iNCoord;                /* number of values in above arrays */

      float fSegmentLength;         /* calculated segment length  */
      float fLineLength;            /* calculated sum of segments */

      int   iElement;               /* Loop counter         */


      /*
       * Get x and y values.
       */
      printf("Line Length Calculator:\n\n");

      do
      {
            printf("Number of x/y values to be entered: ");
            scanf("%d", &iNCoord);

            if(iNCoord < 2) printf("At least two points must be specified\n");

      }while(iNCoord < 2);


      /*
       *    Calculate each segment length and print it out.
```

```
         *    Sum the total line length
         */
        fLineLength = 0.0f;
        for(iElement = 0; iElement < iNCoord-1; ++iElement)
        {
              fSegmentLength =
                    calcHypotenuse( fXCoord[iElement], fYCoord[iElement],
                          fXCoord[iElement+1], fYCoord[iElement+1]);

              fLineLength += fSegmentLength;

              printf("Segment %d\t(%f,%f) to (%f,%f):\t%f\n", iElement+1,
                    fXCoord[iElement], fYCoord[iElement],
                    fXCoord[iElement+1], fYCoord[iElement+1],
                    fSegmentLength);

        }/* End for(i...  */

        /*
         *    Print out the line length
         */
        printf("-----------------------------------\n");
        printf("Length of line: %f\n",fLineLength);

        return 0;

}/* End main()    */


/** FUNCTIONS     */

/*
 * calcHypotenuse:      calculate the distance between two 2D points
 *
 *    inputs:
 *                two floating point coordinates:     x1,y1 and
 *                                                    x2,y2
 *    returns:    a floating point value of the distance between the
 *                input points.
 *
 *    uses: <math.h>
 *
 */
float calcHypotenuse(float fX1, float fY1, float fX2, float fY2)
{
        float fDeltaX;
        float fDeltaY;
        float fHypotenuse;

        /* calculate the differences for X and Y  */
        fDeltaX = fX2 - fX1;
        fDeltaY = fY2 - fY1;

        /* calculate the distance     */
        fHypotenuse = (float) sqrt( fDeltaX*fDeltaX + fDeltaY*fDeltaY );

        return fHypotenuse;

}/* End calcHypotenuse()      */

/** END     */
```