**MTM Standard for C Source Files**

Frank Ackerman

*Software Engineering
Montana Tech of the University of Montana*

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author  | Comment |
| 1.0 | 12/26/10 | Frank Ackerman | Initial version |
| 1.1 | 1/13/11 | Frank Ackerman | Minor changes |

**Montana Tech Software Engineering Students:**

These Montana Tech Method software engineering standards encapsulate Dr. Ackerman’s decades of experience in the software industry, the IEEE software engineering standards, and many suggestions from various texts. They have gone through many revisions and additions over the last several years. They are part of your software engineering studies so that (1) you may have the experience of developing software to a standard (which you may find you need to do if you take a job that requires high reliability software), and so that (2) you will have the experience of developing high quality software. You are also invited to participate in the continuing evolution of these standards by studying them critically and making suggestions for their improvement and correction.

# Purpose

The purpose of this document is to define a Montana Tech CS Department standard for C source files that can be used in courses in Software Engineering or Computer Science that teach C programming or use C programs.

# Introduction

Industry source files tend to have very long lives and to change regularly. Since these files constitute the first level of documentation of the code it is imperative that as well as being syntactically and semantically correct code, they be readily understandable by a defined community of knowledgeable professionals.

This document specifies the requirements that Montana Tech CS Department source files should meet to be readily understood by knowledgeable students and faculty.

# Justification

Some of the requirements in this standard are unusual. They have grown of the author's involvement with software inspections, software reliability, and software quality over decades of industry experience. Creating close to zero defect software is extremely difficult. This standard is based on the belief that a programmer should supply as much information as possible to make his or her code readable and understandable by current and future peers. All code is written using a single, simple fixed width font, sans even such simple devices as italics and underlining. This restricts the expressive clarity of embedded textual material. One of the ways this standard addresses this problem is to require that all variable names not be ordinary words in the natural language of the textual material.

The textual material required by this standard is highly structured to provide just the information essential for succinctly and accurately documenting program requirements, design, testing, and correctness arguments. This standard calls for only a meager number of comments in the code itself – most of the necessary information is given in module and function header comments that use a labeling scheme to link requirements and design elements to blocks of code. This technique encourages the creation of precise, verifiable designs prior to coding[[1]](#footnote-1), facilitates correctness arguments and rigorous inspection, and makes the code itself easier to read and maintain.

Although this standard stands alone, there are companion MTM standards for Java, C++, and C#. Montana Tech computer science and software engineering students take courses in C++, C#, and Java. The Sun Microsystems Coding Conventions for Java are considered by some as an industry standard, so wherever possible this standard is compatible with the Sun document. Some of the major elements it adds to the Sun document that are not related to differences between C and Java are:

* the file heading comment is simplified
* tabs are set every four spaces
* only a few in-code comments are called for
* function header comments immediately precede function headers
	+ - function parameters (except those for main()) are all suffixed PR
		- function header comments may directly cite function parameters and local variables
		- all function headers have a DESCRIPTION section
		- some function headers may also have a REQUIREMENTS and DESIGN section. These are rigidly structured:
			* requirements are most often a single sentence and are identified with a unique label (e.g., R01)[[2]](#footnote-2)
			* design elements must be one of the MTM standard design language constructs
			* blocks of design elements are identified with a unique label (e.g. D01) which must be associated with hanging indented L labels in the code
		- function header comments are set off with dashes except for the header comment for main()
* if and else blocks are set off with braces unless they are a single expression that fits on the same line as the if and else clause. Paired if and else clauses should use the same format.
* else statements start on a separate line
* else if are so written and start on a separate line
* else if statements should only be used for multi-way branching
* Preferable variables are initialized in assignment statements close to where they are first used. However, variables may be initialized where they are declared, but should be in a labeled block of all initialized variables and cited in a design element.
* each line of code should contain at most one statement except in rare cases where the statements are short and the programmer wants to emphasize their relationship
* all closing braces are tagged to identify the element they close
	+ - function definitions are tagged with an in-line comment repeating the name of the function followed by ()
		- if, else, while, do, for, and switch statements are tagged with an in-line comment that at a minimum identifies the construction it closes (e.g., }//if). This may also be followed by a brief comment relating to the semantics of the construction.

# Application

This standard applies to all source files that instructors use as examples and to all source files that students produce as part of a scored assignment.

# Files

This standard applies to the following kinds of source files:

1. main() function .c file
2. other function .c files
3. header .h files

# main() program .c file

The .c file that contains the main program shall have the following parts in the following order:

1. Title block – always present
2. C includes – almost always present (normally <iostream> is included for every program)
3. Application includes – present only if there are application includes
4. Global constant declarations – present only if there are global constants
5. Type definitions – present only if there are type definitions
6. Global variables declarations – present only if there are global variables
7. Function prototypes – present only if there are function prototypes
8. main() header
9. Definition of main()
10. Definition of other functions
11. Source file end comment

Title Block

The title block consists of three C++ comment lines that begin at the left margin. These are the first four lines of the file.

The first line, tabbed a few stops to the right after //, gives the name of the directory/folder that contains this file. Normally this will be the name of the project of which this file is a part. If Visual Suite (VS) is being used, this folder would be created in VS.

The second line, tabbed a few stops to the right after //, gives the name of the file.

The third line gives the current owner's name immediately following //

// FactProjAAF

// factorialFuncAAF.cpp

//Frank Ackerman

If another source was used to construct this file this source should be cited after the owner’s name.

C Includes

A C includes block begins with a blank line. The following is an example.

//-----------

// C includes

//-----------

#include <stdio.h>

Application Includes

An application includes block begins with a blank line. The following is an example.

//---------------------

// Application includes

//---------------------

#include "*appInclude.h*"

Global Constants

A block of global constants begins with a blank line. Global constants are use to give meaningful names to constants that will be used throughout a program. Global constants are written with all caps and underscores. The following is an example.

//-----------------

// Global constants

//-----------------

const long MAX\_FACTOR = 13;

For the most part, global (or local, defined below) constants should be used instead of literals.

Type Definitions

A block of type definitions begins with a blank line. Since we usually define a class in a separate .h file these will usually be enum or struct definitions. Structure names are prefixed with S, enumeration names with E. Names of structure members are suffixed with SV. Names of enumeration constants are written the same way as constants.

//-----------------

// Type definitions

//-----------------

struct Sdate

{

 int daySV;

 int monthSV;

 int yearSV;

};//Sdate

enum ElightColor {RED, GREEN, YELLOW};

Global Variables

A block of global variables begins with a blank lines. Global variables end with the suffix GV. Global variables are never initialized here. The following is an example.

//-----------------

// Global variables

//-----------------

long maxFactGV;

In general, the use of global variables is not encouraged.

Function Prototypes

The function prototypes section begins with a blank lines followed by a section heading:

//---------------------

// Function definitions

//---------------------

Our preferred style is to just list function prototypes before the main program (see next sub-section) and then to supply the function definitions after the main program. This style focuses initial attention on the main, top-level of the program and places the definition of the functions called by main in a subordinate position. Also, this style automatically handles and cross-linked functions.

The rules for function definitions are given below.

Function Prototypes

The function prototypes section begins with one blank line followed by a section heading:

//--------------------

// Function prototypes

//--------------------

All project defined functions are listed here. These functions are given in alphabetical order by function name. The function type is given on a separate line immediately preceding the function name. A blank line separates each function prototype:

double
areaRect(double wdthPR, double lngthPR);

A parameter name must be supplied for each parameter. These names must follow the guidelines for variable names given in the next section. Each parameter name must be suffixed with PR to clearly identify it as a function parameter.

Separate lines may be used to list the parameters:

void

readSalesData(ifstream& infilePR,

 string salesPersonIdsPR[],

 double salesByQuarterPR[][NMBR\_QRTRS\_IN\_YEARS],

 int nmbrOfSalesPersonsPR);

When this is done, placing more than one parameter on a line indicates a relationship, example for a geometric situation lengthPR and widthPR might be placed on the same line.

main() Header Block

A function header block always contains a DESCRIPTION section. It may also contain REQUIREMENTS and DESIGN sections. The main() header block may also contain LOGICAL TEST CONDITIONS, TEST CASES, and CORRECTNESS ARGUMENTS as well, although these are usually more easily constructed in one of the accompanying development documents.

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DESCRIPTION - main

*Brief description of the program for which this is*

*the main function. When there is a spec sheet, this should be an exact copy of the description in the spec sheet.*

REQUIREMENTS

*Rnn labled list of input, output, and processing requirements) for this program. See the section below on Other Functions for further information. If requirements are stated in a separate document this document should be referenced here.[[3]](#footnote-3) Also see the Other Functions section below*

DESIGN

*The overall design of the main function, which must be the*

*overall design of the program. See the Mtech Program Design Language Standard for details. If the design is stated in a separate document this document should be referenced here[[4]](#footnote-4). See the last section of this document for an example.*

LOGICAL TEST CONDITIONS

*See MTM Simple Program Specification Sheet Template.*

TEST CASES

*See MTM Simple Program Specification Sheet Template.*

CORRECTNESS ARGUMENT

*See MTM Simple Program Specification Sheet Template.*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

The heading comment may also contain an optional NOTES section. In this section the items are numbered N01, N02, etc

main() Function Definition

The rest of the main() function block has the outline shown below.

int

main(int argc, char \*args[])

{

 *Declaration of main program variables and constants. See below for*

 *details*

 *Executable program statements. See below for details.*

}//main()

A blank line often precedes the closing brace.

### Declaring function variables and constants

Function variables and constants should all be declared at the beginning of a function before any executable statements. The following guidelines apply.

1. All declaration statements begin at the second tab stop.
2. Except for unusual situations, each variable declaration is made on a separate line.[[5]](#footnote-5) Variables of the same type that are closely related, for example, are attributes of the same object, may sometimes be on the same line to show this close relationship. For example, you might have

 float boxHeight, boxWidth; boxLength; //box dimensions

on the same line.
3. When appropriate, variables of the same type should be grouped together and written in alphabetical order.

int customerName,

 numberOfCustomers,

 supplierName;

1. Variables should be initialized just prior to their first use.
2. Variable names
	1. should be no shorter than three characters.
	2. The index variable in loops is best named explicitly as such, for example ndx, but for short simple loops i, j, k, m, and n may be used. These variables may also be used to match a problem statement
	3. should be descriptive of the data they contain, for example, firstPrime
	4. should not be ordinary English words like prime or first.[[6]](#footnote-6)
	5. may be abbreviated in part or in whole

Abbreviations should be used consistently. The table below list some of the more common abbreviations that have currently been identified.

|  |  |
| --- | --- |
| **Abbreviation** | **Entry** |
| avg | average |
| coef | coefficient |
| cord | coordinate |
| crrnt | current |
| dsply | display |
| elem | element |
| flg | flag |
| frst | first |
| hrs | hours |
| min | minutes |
| ndx | index |
| nmbr | number of (amount) |
| num | Identifing number |
| nxt | next |
| scnd | second |

* 1. always begin in lower case
	2. array variables names should always end in Ary
	3. pointer variable names should always end in Ptr.
	4. should use internal capitals to delineate separate words (i.e., use "camel" casing), for example, lastName rather than last\_name
	5. should, where appropriate, fully qualify the name with the object it is associated with, for example, empLstNm and cusLstNm for "employee last name" and "customer last name".
	6. end in LC for local const constants
	7. end in LS for local static variables
1. Variables that have no other function except to index through a for loop may be declared in the for statement.

### Executable program statements

The statements that make up the body of a program should adhere to the guidelines given below. In addition there are a number of rules for the use and placement of braces. These are covered in a separate section below.

1. No statement begins before the 2nd tab stop. The statement may be preceded by an L-label comment, however. See the last section for an example.
2. Lines that are part of compound statements should be indented a half tab stop (2 spaces)
3. Statement should not line break when they are printed,[[7]](#footnote-7) i.e., the programmer must explicitly break long statements. Subsequent lines of broken statements are indented two spaces unless operator alignment is being used.
4. As just shown, sometimes when breaking on an operator, start the continued line with that operator. In this case, position the initial operator to best show continuity.
5. Blank lines should be used to group logically related statements together.
6. Most terminating (right) braces should be on a line by themselves at the same indentation level as the statement they terminate. Most terminating braces should be tagged with the type of statement they terminate, for example, }//for. Additional information may be included in the tag where appropriate, for example, }//for each employee.[[8]](#footnote-8)
7. Unary operators appear immediately adjacent to their operand.
8. Binary operators are almost always surrounded by spaces and the factor or term enclosed in parentheses.[[9]](#footnote-9)
9. Use of the special else if construction should be reserved for a generalized switch, that is, multi-way branching statements.
10. It is permissible to use a while(true) construction. When using this construction the first statement in the loop should be an if-break statement.
11. Spaces should be used in parenthesized expression to clarify scope, for example

foo(!((canNmbr == empNmbr) && (canName != empName)))

should be written as

foo(!( (canNmbr == empNmbr) && (canName != empName) ))

Other Functions

The block of other functions consists of sub-blocks of other function definitions. These sub-blocks should be ordered alphabetically by function name. Each sub-block should begin with two blank lines. The outline of the text for each function is:

/\*-------------------------------------------------------

DESCRIPTION

*Brief description of what this function does*

REQUIREMENTS

*Rnn labled list of input, output, and processing requirements (if any) for this function unless the function is very small and these requirements are self-evident.*

DESIGN

*The overall design of the function, which must be the*

*overall design of the program. All lines are tabbed over one stop and some lines are preceded by a Dnn number. See the last section of this document for an example. This section may be omitted for very simple functions*

-------------------------------------------------------\*/

*function-type*

*functionName*(*function-parameters*)

{

*Declaration of function variables. See section 5.9.1.*

*Executable program statements. See section 5.9.2*

}//*functionName()*

A blank line may precede the closing brace. The closing right brace is commented with the function name followed by ().

Large, complex functions should contain a DESIGN subsection following the REQUIREMENTS subsection. This function DESIGN should be linked to the code as it is in main(). The heading comment may also contain an optional NOTES section. In this section the items are numbered N01, N02, etc.

REQUIREMENTS

1. REQUIREMENTS should be given unless the function is simple and everything that a user needs to know is given in the DESCRIPTION.
2. Each requirements statement is a statement that can be verified by observation, inspection, or testing. Usually it is a single sentence, although in some cases it may have a list of subordinate clauses. In that case, each clause must be verifiable.
3. Some of the requirements statements must describe the results of executing this function. A function can influence its environment through pointer parameters (or reference parameters), through its return value (if it is not void), through its own static variables (if it has any), and through output to files. Requirements statements should describe any change the function could potentially make to its environment and under what conditions it will make that change. If any functions that this function invokes change the environment these changes must also be referenced.
4. The results of a function can be influenced by its parameters, global variables, its own static variables, and by input from files. These influences should be covered in requirements statements.
5. The requirements should also state any other constraints that should be met by the functions design or implementation.

Source File End Mark

Every source file should end with

//end *filename*

# Comments

By including a keyed, detailed design in the source file we eliminate the need for very many comments in the code. Comments in the code itself should be used sparingly as they can get in the way of clearly reading and understanding the code, especially if they just restate code constructions. Some of the places where it is appropriate to use comments are:

1. // comments describing the purpose or function of a local variable (which is one reason each variable is on a line by itself). These comments should all begin at the same tab stop off to the right
2. comments that add application specific information to the }//if, }//else, and }//while constructions.
3. brief comments placed on top of if or else statements to remind the reader of important aspect of the unfolding calculation

# Rules For Braces

Braces, like parentheses in expressions, should be used liberally. The following rules apply:

1. Function braces

The opening (left) brace for a function definition is on a line by itself at the leftmost margin, as is the closing brace. The closing brace should be immediately followed by a comment tag giving the name of the function.

1. if blocks
	1. always use braces, even for one line ifs unless the statement controlled by the if fits on the same line.
	2. always use the format:

 if (*expression*)

{

 *first statement controlled by* if

 *any additional statements controlled by* if

 }//if *optional descriptive comment*

1. else blocks
	1. always use braces, even for one line else
	2. always use the format:

 else

{

 *first statement controlled by* else

 *any additional statements controlled by else*

 }//else *optional descriptive comment*

1. while blocks
	1. always use braces, even for one line whiles
	2. always use the format:

 while (*expression*)

{

 *first statement controlled by* while

 *any additional statements controlled by* whil*e*

 }//while *optional descriptive comment*

1. do while blocks
	1. always use braces, even for one line whiles
	2. always use the format:

 do

{

 *first statement controlled by* do

 *any additional statements controlled by* do

 } while (*expression*);

1. for blocks
	1. always use braces, even for one line fors
	2. always use the format:

for (initializing-list; *expression*; *altering-list*)

{

 *first statement controlled by* fo*r*

 *any additional statements controlled by* for

 }//for *optional descriptive comment*

1. switch blocks
	1. the case expressions also starts at the same indent level as switch
	2. the body of each case is indented one stop
	3. there is a blank line before each case except the first and a blank line before the closing brace
	4. a default should be included as the last case

 switch (*integral-expression*)

{

 case *value-1*:

 *statement1;*

 *statement2;*

 case *value-2*:

 *statement3;*

 *statement4;*

 default:

 *statement5;*

 }//switch *optional descriptive comment*

Note that switch causes execution to start at a selected case statement (or the default statement) but unless the last statement in a case is a break statement execution continues with the next case.

# Example

// PrintSumThreeNumbers

// printSumThreeNumbers.c

//Frank Ackerman (from Forausan & Gilbert p46)

//----------

//C includes

//----------

#include <stdio.h>

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DESCRIPTION - main

Obtains three intergers from the user and displays their sum

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int

main(int argc, char \*argv[])

{

 char scrnHoldAry[10];

 int aVar,

 bVar,

 cVar,

 sumThree;

/\*L01\*/ printf("\nWelcome. This program adds three integers.\n");

/\*L02\*/ printf("Enter these numbers in the form: nnn nnn nnn \"Enter\"> ");

 scanf("%d %d %d", &aVar, &bVar, &cVar);

/\*L03\*/ sumThree = aVar + bVar + cVar;

/\*L04\*/ printf("The total is %d\n", sumThree);

/\*L05\*/ printf("Thank you. Have a good day.\n\n");

/\*L06\*/ printf("Enter anything to end this program> ");

 scanf("%s", scrnHoldAry);

/\*L07\*/ return 0;

}//main()

//end printSumThreeNumbers.c

# Special Topics

Command Line Arguments

As shown above the special function main() may have the arguments int argc and char \*argv[]. These arguments are given values when the function is invoked by the operating system. When the program with the function main() is invoked from the command line:
 *program string1 string2 ...*
argc is the length of the array argv[], argv[0] has the value *program* and the subsequent values of argv[] are *string1*, *string2*, ... . When any of the string parameters are needed as an integer, say argv[1], the atoi function in <cstdlib> may be used to get the value of the string as an integer:
 int nmbr = atoi(argv[1]);
Visual Studio is fussier about the arguments passed to atoi() so the above will not work. A simple work-around is:
 string tmpStr = argv[1];
 int nmbr = atoi(tmpStr.c\_str());
Of course, in this case <string> must be included.

1. For teaching purposes my criteria for a satisfactory design is that it can be at least abstractly "executed" by a peer other than the author. [↑](#footnote-ref-1)
2. Some requirements naturally break down into sub-requirements. These are labeled a, b, c, etc. [↑](#footnote-ref-2)
3. The use of a separate requirements document violates our stipulation that all textual material be included in the source file. We relax this requirement for student assignments. In a development environment the requirements in the separate document should be copied here before the final version of the code is placed under configuration management. [↑](#footnote-ref-3)
4. See the previous footnote. [↑](#footnote-ref-4)
5. Putting each variable on a line by itself facilitates (a) following the variable by a brief descriptive comment, (b) initializing that variable, (c) changing the type of that variable, (d) organizing variable declarations into meaningful groups. [↑](#footnote-ref-5)
6. Code documentation that is not embedded in source files has a strong tendency of getting lost over time. (Source code files often migrate over many generations of a program. There are files that were created thirty years ago that are still in use today.) This standard emphasizes *in situ* documentation that may include a detailed design in the function heading comment that is directly keyed to the code. These detailed design statements often reference program variables. Since source code is restricted to a single font, names of variables that are English words may lead to confusing and ambiguous English statements. [↑](#footnote-ref-6)
7. The length of a non-breaking line varies with font size and the printer. A good font size is 10. At this size most printer will not break an 80 character line. [↑](#footnote-ref-7)
8. Code is often moved and reformatted. Tagging terminating braces is very helpful in re-establishing visual clues to code logic. It also aids readability generally. [↑](#footnote-ref-8)
9. In some special situations where clarity is enhanced binary operators are not surrounded by spaces and parentheses are not used. [↑](#footnote-ref-9)