**MTM Standard for C# Source Files**

**Version 1.3**May 27, 2012

A. Frank Ackerman

*Software Engineering  
Montana Tech*

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comment |
| 1.0 | 07/16/10 | Frank Ackerman | Initial version |
| 1.1 | 12/12/10 | Frank Ackerman | Update for class exercise |
| 1.2 | 10/02/11 | Frank Ackerman | Updated for ESOF427F11 |
| 1.3 | 05/27/12 | Frank Ackerman | Added reference to XNA in section 1.0 |

**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 the Montana Tech Software Engineering standard for C# source files.

# Introduction

Industry source files tend to have very long lives and to change regularly. Since these files constitute the first level of documentation of a program, it is imperative that source files, as well as being syntactically and semantically correct, be optimally understandable by a defined community of knowledgeable professionals. The community addressed by this standard is the students and faculty taking or teaching software engineering classes offered by the Montana Tech CS Department, especially students majoring in software engineering.

The conventions in *C# Programming Program Design Including Data Structures* by Barbara Doyle are for the most part included in this standard. This standard lays out additional conventions for:

* source file title blocks
* package block
* import headers
* class headers
* class constant, variable, constructor, and method header blocks
* method heading comments:
  + - DESCRIPTION – *<method name>*
      * Mandatory for all functions.
      * A few brief sentences
    - NOTES
      * Additional implementation comments
    - REQUIREMENTS
      * As described in *MTM Simple Program Development Document* section 5.1 – Functional Requirements
    - TEST CONDITIONS
      * As described in *MTM Simple Program Development Document* section 10 – Module Test Conditions
    - TEST DATA
      * As described in *MTM Simple Program Development Document* section 12 – Program Test Data
    - DESIGN
      * As described in *MTM Simple Program Development Document* section 9 – Module Design
    - CORRECTNESS ARGUMENT
      * As described in *MTM Single Module Specification* section 9

Not all methods require headers. If a method does have a header, it must have at least a DESCRIPTION section.

* tagging right braces, and
* naming and tagging variables.

The purpose of these additional conventions is to:

* reduce initial coding errors,
* make it possible to more easily find coding errors during code inspections,
* to reduce ambiguity in function header REQUIREMENTS, DESIGN, NOTES, TEST CONDITIONS, and CORRECTNESS ARGUMENTS
* to enable the construction of CORRECTNESS ARGUMENTS.

This document specifies the requirements that Montana Tech software engineering C# source files should meet to be readily understood by knowledgeable students and faculty taking or teaching Montana Tech software engineering courses. All C# source files created or used in Montana Tech software engineering courses should adhere to this standard.

The MTM software engineering standards include source file standards for C++, Java, C# and XNA. Although specific language or documentation features make it impossible for these standards to be 100% consistent, wherever possible unnecessary inconsistencies have been eliminated.

# 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 software engineering assignment.

Two types of C# source files (.cs) are distinguished:

* those that contain the Main() entry point, and
* those that do not, that is, they just define a class whose objects may be instantiated elsewhere, and whose methods must be invoked from elsewhere.

For the first type of .cs file the first method in the class should be:

public static void  
 Main(string[] args)   
 {  
 ....  
 }//Main()

The string[] args argument may be elided if the application does not process command line parameters.

The class may be given a name appropriate to the application, or generically, Main, so that the name of the file is Main.cs.

In this version of this standard for any application specific classes, say *AppClass*, referenced in the Main class, there must be a corresponding *AppClass.*cs file.

# .cs File For Class Containing Main()

The .cs file that defines the class that contains the Main() method shall have the following parts in the following order:

1. Title block – always present
2. C# namespace header comment block
3. C# namespace statements
4. Class header comment block
5. Class definition header
6. Class member constants comment block
7. Class member constants (or //none)
8. Class member objects/variables comment block
9. Class member objects/variables (or //none)
10. Class methods/functions comment block
11. Main() function header comment block
12. Definition of Main() method.
13. Definition of other class methods
14. Closing brace for class definition
15. Closing brace for application name space
16. //end *fileName*

Typically a class containing a Main function is not explicitly instantiated and thus all member functions called from Main must be declared **static**.

The Simple.java file displayed below illustrates some of these points:

|  |  |
| --- | --- |
| (1) | // HelloWorldAAF  // HelloWorldAAF.cs  //Frank Ackerman (from B. Doyle, C# Programming |
| (2,3) | //-------------  //C# namespaces  //-------------  using System; |
| (4,5) | namespace FirstProgram  { |
| (6) | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Textbook example to show parts of a .cs file that contains Main()  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |
| (7) | public class HelloWorld  { |
| 12 | //-------------  //Member Method  //------------- |
| 13 | /\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* DESCRIPTION - Main  \* Textbook illustrative program.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |
| 14 | public static void  Main()  {  Console.WriteLine("Hello World");  }//Main() |
| 16 | }//class HelloWorld |
| 17 | }//namespace FirstProgram |
| 18 | //end HelloWorldAAF.cs |

Title Block

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

The first line, tabbed a few stops to the right after //, gives the name of the application to which this file belongs. 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 //. If this program is an adaptation of a program written by someone else, attribution should be made here

// HelloWorldAAF

// HelloWorld.cs

//Frank Ackerman

C# namespaces header comment block

A Java import block begins with a blank line. The following is an example.

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

//C# namespaces

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

C# namespaces

Listed one pre line immediately below the imports header comment block. This block is not present in the unusual case that this file does not reference any other C# namespaces

Class header comment block

A class header comment block begins with a blank line. The following is an example:

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

\* Textbook example to show parts of a .java file

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

The format for all C# block comments is slightly different than those for C++ and Java. The format shown above is the default one used by the Visual Studio C# source editor.

Class definition header

In the following schema notice:

* that the class definition header begins immediately after the class header comment
* that class definitions begins in column 1
* the brace that closes the class definition starts in column 1 and is commented with the name of the class.

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

\* Textbook example to show parts of a .java file

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

public

class HelloWorld

{

...

}//class HelloWorld

or

public

class HelloWorld {

...

}//class HelloWorld

Class member constants comment block

A class constants comment block begins with a blank line. The following is an example

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

//Member Constats

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

Class member constants

Listed one pre line immediately below the constants comment block.

Class member objects comment block

A class objects comment block begins with a blank line. The following is an example

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

//Member Object(s)

//---------------\_

Class member objects

Listed one pre line immediately below the objects comment block

Class methods comment block

A class methods comment block begins with a blank line. The following is an example.

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

//Member Method(s)

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

Main() method header

The Main() method header begins with a blank line. The following is an example.

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

\* DESCRIPTION - Main

\* *Brief description of this application or reference to* \* a *Simple Program Specification Sheet or Document*

\*

\* NOTES

\* *An optional section providing additional information or*  
 \* *explanation*.

\*

\* REQUIREMENTS

\* *Application requirements statement or reference to a* \* *Simple Program Specification Sheet or Document*

\*

\* TEST CONDITIONS

\* *Test conditions or reference to a*

*\* Simple Program Specification Sheet or Document*

\*

\* TEST DATA

\* *Test c or reference to a*

*\* Simple Program Specification Sheet or Document*

\*

\* DESIGN

\* *Main() design or reference to a*

*\* Simple Program Specification Sheet or Document*

\*

\* CORRECTNESS ARGUMENT *(optional)*

\* *Correctness Argument or reference to a*

*\* Simple Program Specification Sheet or Document*

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

Definition of Main() method

The definition of the Main() method immediately follows the Main() header. Notice

* that method attributes are give on a separate line immediately above the function signature
* the parameters for Main(,) if present, are always string[] args
* that tabs are set at 4, 8, 12, ...
* that method code begins at the 3rd tab stop
* that closing right braces are always commented to indicate the item being closed

public static void  
Main (string args[])

{

*Method Body:*

*Code for Main() leading label linked to DESIGN in the header comment. See Section 6 below*

}//Main()

O*ther class methods*

}//class *ClassName*

or[[1]](#footnote-1)

Main (string args[]) {

*Method Body:*

*Code for Main() leading label linked to DESIGN in the header comment. See Section 6 below*

}//Main()

Definition of other class methods

Other class methods are defined similarly to the Main() method except these methods will not contain a TEST CONDITIONS or TEST DATA section.

For all methods with parameters (except Main()) the names of the parameters are suffixed with PR.

Closing brace for class definition

The class closing brace should be preceded by a blank line. The brace includes a //class comment naming the class.

# cs File for Any Other Class

The .cs file that defines any other class is similar to that for the class that contains Main() with the following additions (numbered):

* Title block – always present
* C# namespace header comment block
* C# namespace using statements
* Application namespace header comment block
* Application namespace
* Class header comment block
* Class definition header
* Class member constants comment block
* Class member constants
* Class member objects comment block
* Class member objects
* Class constructors comment block

1. Class constructor definitions.

* Class methods/functions comment block
* Method header block
* Definition of method.
* Headers and definitions of any other methods in this class

1. Definition of any private classes

* Class definition closing brace

Definitions of private classes may appear after class member objects, but should not appear both places.

The .cs file below is an example. Notice:

* that attribution is given in the file header since this code was derived from the source cited
* that method code is indented enough to provide space for a 7 character label
* that empty blocks (labeled //none below) need not be present.

// KiloConverter

// KiloConverterWindow.cs

//Frank Ackerman from Java standard example of same name

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

//C# namespaces

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

using System;

using System.Windows.Forms;

namespace KiloConverterCsharp

{

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

\* The KiloConverterWindow class displays a System.Windows.Forms.Form

\* that lets the user enter a distance in kilometers. When a

\* Calculate button is clicked the distance is miles is displayed.

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

public partial

class KiloConverterWindow : Form

{

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

//Member constants

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

//none

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

//Member Objects

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

double kilosMO,

milesMO;

string kilosInMO;

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

//Class construtor

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

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

\* DESCRIPTION - KiloConverterWindow

\* The no-arg constructor for the KiloConverterWindow class.

\*

\* REQUIREMENTS

\* R01 \*\*Constraints

\* none

\*

\* R02 \*\*Preconditions

\* none

\*

\* \*\*Invocation

\* R02 Automatically when a new KiloConverterWindow is instantiated.

\*

\* \*\*Input/Output

\* R03 The KiloConverterWindow user interface is a single user window.

\* R04 The title of the user window is "Kilometer Converter."

\* R05 The Kilometer Converter window has a Close button.

\* R06 The Kilometer Converter window contains:

\* (a) a labeled text box, enterKilosTxt, for entering a

\* string of digits representing kilometers that are

\* to be converted to miles.

\* (b) a "Calculate" bttn to invoke the conversion from kilometers

\* to miles and dsply the result to 4 decimal places in

\* another labeled textbox, displayMilesTxt

\* R07 The Kilometer Converter window application is terminated by

\* clicking the window Close button.

\*

\* R08 If the user enters other than digits in the text box then when the

\* Calculate button is clicked the result is undefined.

\*

\* R09 \*\*Postconditions

\* none

\*

\* \*\*Use Cases

\* R10 Entering 10.0 in the enterKilosTxt text box and clicking the

\* Calculate button will display 6.2140 in the displayMilesTxt text

\* box.

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

public KiloConverterWindow()

{

InitializeComponent();

}//KiloConverterWindow() no-arg constructor

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

//Member Method

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

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

\* DESCRIPTION - calcBtn\_Click

\* This method executes when the user clicks the

\* Kilometer Converter window Calculate button.

\*

\* REQUIREMENTS

\* R01 \*\*Constraints

\* none

\*

\* \*\*Preconditions

\* R02 A KiloConverterWindow (see above definition) object has been

\* instantiated.

\*

\* R03 User has entered digits and only digits in enterKilosTxt.

\*

\* \*\*Invocation

\* R04 This method is invoked when user clicks Kilometer Converter window

\* Calculate button.

\*

\* \*\*Input/Output

\* R05 The displayMilesTxt text box displays the value of

\* enterKilosTxt in miles.

\*

\* R06 \*\*Postconditions

\* none

\*

\* \*\*Use Cases

\* R05 When the user enters 1 kilometer the miles displayed is

\* 0.6214.

\*

\* DESIGN

\* D01 Get user entered text from enterKilosTxt;

\* D02 Calculate equivalent miles from user entered text;

\* D03 Display miles in displayMilesTxt.

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

private void

calcBtn\_Click(object sender, EventArgs e)

{

const double KILO\_TO\_MILES = 0.6214;

/\*L01\*/ kilosInMO = enterKilosTxt.Text;

kilosMO = double.Parse(kilosInMO);

/\*L02\*/ milesMO = kilosMO \* KILO\_TO\_MILES;

/\*L03\*/ displayMilesTxt.Text = String.Format("{0:F4}", milesMO);

}//calcBtn\_Click()

}//class KiloConverterWindow

}//namespace KiloConverterCsharp

//end KiloConverterWindow.cs

Class constructors comment block

A class constructors comment block begins with a blank line. The following is an example

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

//Constructor(s)

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

Class constructor definitions

Constructors follow the same standards as other methods except

* 1. they must have the same name as the class,
  2. they must be public, and
  3. they cannot have a type (even void).

Each constructor must have a different signature.

Private class definitions

All the rules given herein apply to nested classes defined within a public class except for additional tabs to delineate hierarchy.

# Method Heading

Method names must begin with an upper case letter. Except for Main, method names should be an action verb phrase that briefly describes what the method is intending to accomplish. A method name should not be an English word. Subsequent words that are part of the method name should begin with an upper case letter, that is, “camel casing” should be used.

# Method Body

A method body is all of the code that appears between the opening left brace in a method definition and the corresponding closing right brace. See section 4.14 for an example of the context for a method body.

The top level constructs in a method body must be tabbed over 2 stops from the method definition header.

A method body consists of:

1. method constant definitions
2. method variable declarations
3. method code

## Method Constant Definitions

Method constant (consts) definitions should be listed before any method variables with a space separating constant definitions from variable declarations. The rules for constant definitions are the same as the rules given below for variable declarations except constants are named using only upper case letters (and digits) with parts of the name separated by underscores.

## Method Variable Declarations

Method variables should all (except for special cases where this is not appropriate) be declared at the beginning of a method before any executable statements. The following guidelines apply.

1. Variable types should be listed in alphabetical order.
2. Each variable type should appear only once with all the variables of that type listed alphabetically on separate lines separated by commas (except see the next paragraph).
3. Except for unusual situations, each variable declaration is made on a separate line.[[2]](#footnote-2) 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.
4. Except where required for dynamically allocated variables, variables should not be initialized in a declaration statement. Such initialization is part of the program logic and should be called out in a design statement that has a corresponding label in the code. Standard i/o objects may be initialized when they are defined.
5. Variable names
   1. should always begin with a lower case letter
   2. should always consists of at least three characters (but loop indexes may use i, j, k, m, and n)
   3. should be descriptive of the data they contain, for example, firstPrime
   4. should not be ordinary English words like prime or first.[[3]](#footnote-3)
   5. may be abbreviated by eliding vowels (vowels are sometimes helpful for clarity)
   6. should use internal upper case to delineate separate words, for example, lastName rather than last\_name
   7. should, where appropriate, fully qualify the name with the object it is associated with, for example, employeeLastName.
6. Variables that must be initialized when they are declared should be listed separately at the end of the initial group of declaration statements and, when L lables are being used, given the label L*nn* to match a variable initialization statement in the DESIGN. Variable initialization is an important part of the logic of a program.
7. Variables that have no other function except to index through a for loop may be declared in the for statement.
8. The above rules apply to naming method parameters and class member variables except:
   1. method parameters should always be suffixed with PR, and
   2. defined member objects should always be suffixed with MO.

## Method Code

### 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. Statement blocks that are part of compound statements must be indented one additional tab stop.
2. Statement should not line break when they are printed,[[4]](#footnote-4) i.e., the programmer must explicitly break long statements. Subsequent lines of broken statements are indented two spaces unless operator alignment is being used.
3. As just shown, when breaking on an operator, start the continued line with that operator. In this case, position the initial operator to best show continuity.
4. Blank lines should be used to group logically related statements together.
5. Most terminating (right) braces should be on a line by themselves at the same indentation level as the statement they terminate. 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.[[5]](#footnote-5)
6. Unary operators appear immediately adjacent to their operand.
7. Binary operators are almost always surrounded by spaces and the factor or term enclosed in parentheses.[[6]](#footnote-6)
8. Use of the special else if construction should be reserved for a generalized switch, that is, multi-way branching statements.
9. 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.
10. 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) ))

# Rules For Braces

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

1. if blocks
   1. always use braces, even for one line ifs unless the entire construction fits on one line and else it not used.
   2. always use the format:

if (*expression*)

{

*first statement controlled by* if

*any additional statements controlled by* if

}//if *optional descriptive comment*

or

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*

or

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*

or

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*);

or

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*

or

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 and a blank line before the closing brace

switch (*switchable-expression)*

{

case *value-1*:

*statement1;*

*statement2;*

case *value-2*:

*statement3;*

*statement4;*

default:

*statement5;*

}//switch *optional descriptive comment*

or

switch (*switchable-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.

1. This standard supports both of the common styles of bracing. However, the same style must be used throughout a file, and preferably, throughout an application. [↑](#footnote-ref-1)
2. 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-2)
3. 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, many generations of a program. There are source files that were created forty 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-3)
4. The length of a non-breaking line varies with font size and the printer. A good font size is 10. At this size most printers will not break an 80 character line. [↑](#footnote-ref-4)
5. 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-5)
6. In some special situations where clarity is enhanced binary operators are not surrounded by spaces and parentheses are not used. [↑](#footnote-ref-6)