# Coding Guidelines

## Braces

|  |
| --- |
| 1. Put braces on their own line (especially if there is code between them) (SC) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  {  int value;  cin >> value;  cout << value + 4;  } | |
| |  |  | | --- | --- | | **Bad** | for (int i = 0; i < 10; i++) {  int value;  cin >> value;  cout << value + 4;  } | |

|  |
| --- |
| 1. Be consistent with braces for linked code sections (G) |
| |  |  | | --- | --- | | **Good** | if (1 != 2)  cout << “true”;  else  cout << “false”;  if (1 != 2)  {  int a;  cin >> a;  }  else  {  cout << “false”;  } | |
| |  |  | | --- | --- | | **Bad** | if (1 != 2)  {  int a;  cin >> a;  }  else  cout << “false”; | |

|  |
| --- |
| 1. A code block consisting of two or more lines should be surrounded by braces (S)   A single line should not be surrounded |
| |  |  | | --- | --- | | **Good** | int i;  for (i = 0; i < 10; i++)  cout << i;  if (i != 10)  {  i = 10;  cout << “An error occurred.”;  } | |
| |  |  | | --- | --- | | **Bad** | int i;  for (i = 0; i < 10; i++)  {  cout << i;  }  While (i != 20)  if (i < 20)  i++; | |

## Comments

|  |
| --- |
| 1. Put comments on their own line (G) |
| |  |  | | --- | --- | | **Good** | // This is a good comment  int i = 12; | |
| |  |  | | --- | --- | | **Bad** | int I = 12; // This is a bad comment | |

|  |
| --- |
| 1. Comments should be directly followed by a line of code (G) |
| |  |  | | --- | --- | | **Good** | int i = 12;  // I’m doing OK  i++; | |
| |  |  | | --- | --- | | **Bad** | int i = 12;  // I’m so lonely :(  i++; | |

|  |
| --- |
| 1. Comments should be preceded by an empty line (G) |
| |  |  | | --- | --- | | **Good** | // The stuff  void\* stuff;  // And the things  Void\* things; | |
| |  |  | | --- | --- | | **Bad** | // The stuff  void\* stuff;  // And the things  Void\* things; | |

|  |
| --- |
| 1. Avoid unnecessary comments (G)   If it’s pretty obvious what the code does, comments are redundant |
| |  |  | | --- | --- | | **Good** | // The following line resolves an issue where a collection was  // too large  myCollection.where(i => i > 5).limit(20).toList(); | |
| |  |  | | --- | --- | | **Bad** | // Increment i  i++; | |

## Naming

|  |
| --- |
| 1. Avoid confusing names (G) |
| |  |  | | --- | --- | | **Good** | int byteCounter;  string firstName, lastName; | |
| |  |  | | --- | --- | | **Bad** | int a, f, bis, pw, d; | |

|  |
| --- |
| 1. Don’t use Hungarian notation (S) |
| |  |  | | --- | --- | | **Good** | string firstName, lastName;  bool hasAdminPrivileges; | |
| |  |  | | --- | --- | | **Bad** | string strFirstName, strLastName;  bool bHasAdminPrivileges; | |

|  |
| --- |
| 1. Use camel case (S) |
| |  |  | | --- | --- | | **Good** | string theStuff, theThings;  bool jelloPuddingPops; | |
| |  |  | | --- | --- | | **Bad** | string the\_stuff, the\_things;  bool jellopuddingpops; | |

|  |
| --- |
| 1. Use “m\_” for member variables (G) |
| |  |  | | --- | --- | | **Good** | class foo  {  bool m\_bar;  }; | |
| |  |  | | --- | --- | | **Bad** | class foo  {  bool bar;  }; | |

|  |
| --- |
| 1. Member and local variables start with a lowercase letter (G) |
| |  |  | | --- | --- | | **Good** | class foo  {  foo()  {  m\_bar = 0;  for (int i = 0; i < 10; i++)  cout << m\_bar++;  }  int m\_bar;  }; | |
| |  |  | | --- | --- | | **Bad** | class foo  {  foo()  {  m\_Bar = 0;  for (int I = 0; I < 10; I++)  cout << m\_Bar++;  }  int m\_Bar;  }; | |

|  |
| --- |
| 1. In C#, start namespace, class, method, and property names with uppercase letters (G) |
| |  |  | | --- | --- | | **Good** | namespace Foobar  {  public class Foo  {  public int Bar { get; private set; }  public Foo() { Bar = 0; }  }  } | |
| |  |  | | --- | --- | | **Bad** | namespace foobar  {  public class foo  {  public int bar { get; private set; }  public foo() { bar = 0; }  }  } | |

## Organization

|  |
| --- |
| 1. In C#, sort #using statements alphabetically (G) |
| |  |  | | --- | --- | | **Good** | #using Microsoft.Xna.Framework;  #using Microsoft.Xna.Framework.Audio;  #using Microsoft.Xna.Framework.Content;  #using System;  #using System.Collections.Generic; | |
| |  |  | | --- | --- | | **Bad** | #using Microsoft.Xna.Framework.Content;  #using System.Collections.Generic;  #using System;  #using Microsoft.Xna.Framework.Audio;  #using Microsoft.Xna.Framework; | |

|  |
| --- |
| 1. One class per file (G) |
| |  |  | | --- | --- | | **Good** | #using System;  ...  namespace Foo  {  public class Bar  {  ...  } } | |
| |  |  | | --- | --- | | **Bad** | #using System;  ...  namespace Foo  {  public class Bar  {  ...  }  public class Bar2  {  ...  } } | |

|  |
| --- |
| 1. Group common files in a folder (G) |

|  |
| --- |
| 1. All files in a folder should be in a single, unique namespace (G) |

|  |
| --- |
| 1. Name a file after the class it contains (G) |
| |  |  | | --- | --- | | **Good** | // Bar.cs  namespace Foo  {  public class Bar  {  ...  }  } | |

|  |
| --- |
| 1. Sort primarily by public properties, methods and variables at the top, with private at the bottom (G) |
| |  |  | | --- | --- | | **Good** | public class Foo  {  public Foo() {}  public bool BrianIsCool() { return m\_mmhmm; }  string FirstName { get; set; }  string LastName { get; set; }  const bool m\_mmhmm = true;  } | |
| |  |  | | --- | --- | | **Bad** | public class Foo  {  string FirstName { get; set; }  public Foo() {}  const bool m\_mmhmm = true;  public bool BrianIsCool() { return m\_mmhmm; }  string LastName { get; set; }  } | |

|  |
| --- |
| 1. Sort secondarily by type of object (G)   Properties at the top, then constructors, then methods, then variables |
| |  |  | | --- | --- | | **Good** | public class Foo  {  public string FirstName { get; set; }  public string LastName { get; set; }  public Foo() {}  public bool BrianIsCool() { return m\_mmhmm; }  public const bool m\_mmhmm = true;  } | |
| |  |  | | --- | --- | | **Bad** | public class Foo  {  public string FirstName { get; set; }  public Foo() {}  public const bool m\_mmhmm = true;  public bool BrianIsCool() { return m\_mmhmm; }  public string LastName { get; set; }  } | |

## Spacing

|  |
| --- |
| 1. Use tabs instead of spaces (S) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  cout << i; | |
| |  |  | | --- | --- | | **Bad** | for (int i = 0; i < 10; i++)  cout << i; | |

|  |
| --- |
| 1. Omit extra spaces within a line (SC) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  cout << system.(“pause”); | |
| |  |  | | --- | --- | | **Bad** | for ( int i = 0; i < 10; i++ )  cout << system.( “pause” ); | |

|  |
| --- |
| 1. Add empty spaces between code “paragraphs” (G) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  {  string temp;  cin >> temp;  for (int j = 0; j < temp.size() / 2; j++)  swap(temp[j], temp[temp.length() – j – 1]);  cout << temp;  } | |
| |  |  | | --- | --- | | **Bad** | for (int i = 0; i < 10; i++)  {  string temp;  cin >> temp;  for (int j = 0; j < temp.size() / 2; j++)  swap(temp[j], temp[temp.length() – j – 1]);  cout << temp;  } | |

|  |
| --- |
| 1. Don’t begin or end a file with empty lines (G) |

|  |
| --- |
| 1. Indent code blocks once (and only once) (SC) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  {  string temp;  cin >> temp;  for (int j = 0; j < temp.size() / 2; j++)  swap(temp[j], temp[temp.length() – j – 1]);  cout << temp;  } | |
| |  |  | | --- | --- | | **Bad** | for (int i = 0; i < 10; i++)  {  string temp;  cin >> temp;  for (int j = 0; j < temp.size() / 2; j++)  swap(temp[j], temp[temp.length() – j – 1]);  cout << temp;  } | |

|  |
| --- |
| 1. Add a single space after a keyword and before parentheses (SC) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  {  if (i % 2 == 0)  cout << i;  } | |
| |  |  | | --- | --- | | **Bad** | for(int i = 0; i < 10; i++)  {  if(i % 2 == 0)  cout << i;  } | |

|  |
| --- |
| 1. Avoid operator clutter (SC)   Also watch clutter around semi-colons and parentheses |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  {  if (i % 2 == 0)  cout << i;  } | |
| |  |  | | --- | --- | | **Bad** | for (int i=0;i<10;i++)  {  if (i%2==0)  cout<<i;  } | |

|  |
| --- |
| 1. Don’t try to line up values (SC) |
| |  |  | | --- | --- | | **Good** | int one = 1;  int two = 2;  int three = 3;  int four = 4;  int five = 5;  int six = 6;  int seven = 7; | |
| |  |  | | --- | --- | | **Bad** | int one = 1;  int two = 2;  int three = 3;  int four = 4;  int five = 5;  int six = 6;  int seven = 7; | |

|  |
| --- |
| 1. Omit extra empty lines (S) |
| |  |  | | --- | --- | | **Good** | for (int i = 0; i < 10; i++)  {  string temp;  cin >> temp;  for (int j = 0; j < temp.size() / 2; j++)  swap(temp[j], temp[temp.length() – j – 1]);  cout << temp;  } | |
| |  |  | | --- | --- | | **Bad** | for (int i = 0; i < 10; i++)  {  string temp;  cin >> temp;  for (int j = 0; j < temp.size() / 2; j++)  swap(temp[j], temp[temp.length() – j – 1]);  cout << temp;  } | |

|  |
| --- |
| 1. In C#, auto-properties should be all on one line. (S) |
| |  |  | | --- | --- | | **Good** | public class Foo  {  public string FirstName { get; set; }  public string LastName { get; set; }  } | |
| |  |  | | --- | --- | | **Bad** | public class Foo  {  public string FirstName  {  get;  set;  }  public string LastName  {  get; set;  }  } | |

## Miscellaneous

|  |
| --- |
| 1. In C#, avoid unnecessary #using statements (remove and sort) (G) |

|  |
| --- |
| 1. Minimize return points as much as possible (G) |

|  |
| --- |
| 1. Catch/Throw as many exceptions as possible (within reason) (G)   Consider every possibility |