Coding Standards for C++

Version 5.0

Why have coding standards?

It is a known fact that 80% of the lifetime cost of a piece of software goes to maintenance. Therefore it makes sense for all programs within an organization to be as consistent as possible. Code conventions also improve the readability of the software.

This document specifies the coding standards for all computer science courses using C++ at Pacific University. It is important for you to adhere to these standards in order to receive full credit on your assignments.

The document is divided into four main sections:

- Naming Conventions
- Formatting
- Comments
- Printing

Naming Conventions

Constants

A constant is to be mnemonically defined using all capital letters and underscore characters such as MAX_NAME_CHARS. Separate words using an underscore character. Further, your program is to contain no "magic constants." That is, all magic constants must be declared **const** to make program modification easier. In the case below, 100 is a magic constant and if used in several places throughout a program, can create problems if 100 is to be modified for any reason.

Poor Program Style

```
ins.open ("message.dat");
.....
for (indx = 0; indx < 100; ++indx)
{
.....
}
```

Correct Program Style

```
const int MAX_GRADE_SCORES = 100;
const char INPUT_FILE[] = "scores.dat";

ins.open (INPUT_FILE);
.....
for (indx = 0; indx < MAX_GRADE_SCORES; ++indx)
{
.....
}
```

Notice: Constants like 0 and 1 are usually acceptable unless they represent values such as true and false in which case they should be declared as constants.

Variable Names

1) A variable name is defined in all lowercase letters unless the variable name contains multiple names such as readStudentRecord. After the first word, each subsequent word has the first letter capitalized with the remainder of the word made up of lowercase letters.

2) Variable names are to be mnemonic unless the variable is being used in a for loop in which case the names i, j, k, l, m, n are acceptable names to be used. If however the nested loop is being used in conjunction with a two-dimensional array, then the names row and column should be used.

3) Global variables must begin with g so that a name such as gHashTable denotes a global variable.

4) To aid in identifying the type of a variable, we will use the following prefixes.

Type Indicator is a	**Text Prefix**	**Variable Name Example**
boolean	b	`bFlag`
pointer	p	`char *pName`
reference	r	`char &rSSNum`
handle	h	`void **hWindow`
null terminated string	sz	`char szFileName[10]`
structure	s	`Home sPerson`
class	c	`Identity cPerson`
globals	g	`char gNumFiles`

Poor Program Style

```
int L (char n[])
{
  for (int i = 0; n[i] != '\0'; ++i);

  return i;
}
```

Good Program Style

```
int strLength (char name[])
{
  int count;

  for (count = 0; '\0' != name[count]; ++count)
  {
  }

  return count;
}
```

Class, Struct, and Union Names

Class, Struct, and Union definitions will follow the regular variable naming conventions except the first letter of the class or struct must be capitalized. Further, class and struct definitions are to exist in a header file (.h file) associated with the .cpp source file associated with the project.

Note: Struct definitions do not begin with an s and class definitions do not begin with a c. Only variable declarations begin with an s or c.

Poor Program Style for Structs

```
struct t
{
  int d;
  int h;
  int m;
  int s;
};
```

Good Program Style for Structs

```
struct Time
{
  int days;
  int hours;
  int minutes;
  int seconds;
};
```

Poor Program Style for Classes

```
class rat
{
  public:
    rat ();
    rat (int, int);
    setvalues (int, int);

  private:
    int n;
    int d;
};
```

Good Program Style for Classes

```
class Rational
{
  public:
    Rational ();
    Rational (int, int);
    int getNumerator ();
    int getDenominator ();
    void setNumerator (int);
    void setDenominator (int);

  private:
    int numerator;
    int denominator;
};
```

Method Name - methods are named using the standard naming convention described for variables where the first word begins with a lowercase letter and each subsequent word has the beginning letter capitalized. There is no need to document a method prototype whose function is clear from the name; however, a method whose function is not clear from the name must be documented properly.

Class Implementation

Classes need to be implemented using two files. The first file is a .h file that contains the definition of the class. The second file is a .cpp file that contains the actual implementation of the methods included in the class definition. The .cpp file includes the .h file at the top of the file.

Rational Class and Implementation Example

```
//***************************************************************************
// File name:  rational.h
// Author:     Joe Bloggs
// Date:       1/15/08
// Class:      CS250
// Assignment: Rational
// Purpose:    To define the header file for the rational module
//***************************************************************************

#ifndef RATIONAL_H
#define RATIONAL_H

class Rational
{
public:
      Rational (int, int);
      void setNumerator (int);
      void setDenominator (int);
      int getNumerator () const;
      int getDenominator () const;
      void print ()  const;
      void setValues (int, int);
      bool equal (const Rational &) const;
      Rational multiply (const Rational &);

private:
      int numerator;
      int denominator;
};

#endif
```

```
//***************************************************************************
// File name:  rational.cpp
// Author:     Joe Bloggs
// Date:       1/15/08
// Class:      CS250
// Assignment: Rational
// Purpose:    Implements constructors & methods of class Rational
//***************************************************************************

#include "stdafx.h"
#include <iostream>
#include "Rational.h"

using namespace std;

//***************************************************************************
// Constructor: Rational
//
// Description: Initializes data members to default values
//
// Parameters:  None
//
// Returned:    None
//***************************************************************************

Rational::Rational (int numerator = 0, int denominator = 1)
{
  setNumerator (numerator);
  setDenominator (denominator);
}

//***************************************************************************
// Method:      setNumerator
//
// Description: Changes the value of the numerator to the value input.
//
// Parameters:  numerator   - numerator of the fraction
//
// Returned:    None
//***************************************************************************


void Rational::setNumerator (int numerator)
{
  this->numerator = numerator;
}
```

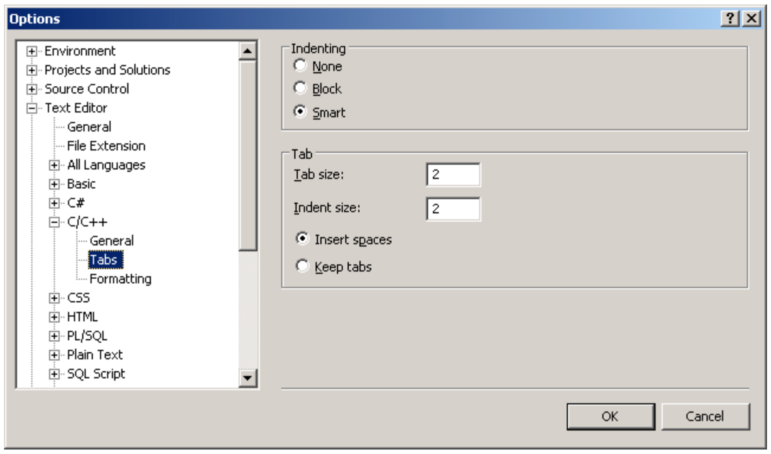
```
//***************************************************************************
// Method:      setDenominator
//
// Description: Changes the value of the denominator to the value input.
//
// Parameters:  denominator   - denominator of the fraction
//
// Returned:    None
//***************************************************************************


void Rational::setDenominator (int denominator)
{
  this-> denominator = denominator;
}

//***************************************************************************
// Method:      getNumerator
//
// Description: Returns the value of the numerator.
//
// Parameters:  None
//
// Returned:    The numerator value
//***************************************************************************


int Rational::getNumerator () const
{
  return this->numerator;
}

//***************************************************************************
// Method:      getDenominator
//
// Description: Returns the value of the denominator.
//
// Parameters:  None
//
// Returned:    The denominator value
//***************************************************************************


int Rational::getDenominator () const
{
  return this->denominator;
}
```

```
//***************************************************************************
// Method:      print
//
// Description: Outputs a fraction in the form numerator / denominator to the
//              screen
//
// Parameters:  None
//
// Returned:    None
//***************************************************************************


void Rational::print () const
{
  cout << " " << getNumerator() << "/" << getDenominator() << " " << endl;
}

//***************************************************************************
// Method:      equal
//
// Description: Compares two objects of Rational returning a value of true if
//              the numerators and denominators of both fractions are the
//              same.
//
// Parameters:  fraction - object of type Rational.
//
// Returned:    true if objects are equal; else, false
//***************************************************************************


bool Rational::equal (const Rational &fraction)  const
{
  return (numerator == fraction.numerator &&
          denominator == fraction.denominator);
}

//***************************************************************************
// Method:      multiply
//
// Description: Multiples the numerators and denominators of two objects.
//
// Parameters:  fraction - object of type Rational.
//
// Returned:    An object of type Rational that contains the result of the
//              multiplication.
//***************************************************************************

Rational Rational::multiply (const Rational &fraction) const
{
  Rational tempFraction (0, 0);

  tempFraction.numerator = numerator * fraction.numerator;
  tempFraction.denominator = denominator * fraction.denominator;

  return tempFraction;
}
```

Formatting

Indentation

Two spaces must be used as the unit of indentation per tab. Every IDE (Integrated Development Environment) such as Visual Studio includes an option for changing the number of spaces in a tab. These can usually be found in the preferences section. In Visual Studio 2005 go to Tools->Options which brings up the picture below. Then select Text Editor->C/C++->Tabs. At this point make sure the Tab Size and Indent Size are 2 and that the radio button for insert spaces is selected. Select these options before typing in any of your code.



Indenting
None
Block
[✓ Smart]

Tab
Tab size: 2
Indent size: 2
[✓ Insert spaces]
Keep tabs

Line Length

Lines must be no longer than 78 characters. Anything longer than 80 characters is normally not handled well in many terminals and tools.

Wrapping Lines

If an expression cannot fit on a single line then break it:

- After a comma
- Before an operator

Make sure that the new line is aligned with the beginning of the expression at the same level on the previous line.

Spaces

All arithmetic and logical operators must have one space before and after the operator. The only exceptions are:

- Unary operators
- The period
- No spaces before the comma and only one space after the comma

Blank Lines

Use blank lines to separate distinct pieces of code. For example, one blank line before and after a while loop helps the code reader. The important thing to remember is that blank lines must be used consistently.

Braces

Any curly braces that you use in your program (e.g. surrounding classes, functions) must appear on their own lines. Any code within the braces must be indented relative to the braces.

```
class User
{
public:
      User();

private:
      char firstName[MAXNAMESIZE];
      char lastName[MAXNAMESIZE];
};
```

Comments

Comments should be used to explain the purpose of the code fragment they are grouped with. Comments should state what the code is doing, while the code itself shows how you are doing it.

Use comments sparingly and only comment code segments that are not obvious. Giving your variables meaningful names will improve the readability of your code and reduce the need for comments.

File Header

The main purpose of a file header is to explain the purpose of the program as briefly as possible. You must include the following sections in your program header:

- File name
- Your name
- Date
- Class Title
- Assignment Title
- Purpose

```
//****************************************************************************
// File name:  main.c
// Author:     Joe Bloggs
// Date:       1/15/08
// Class:      CS250
// Assignment: Rational
// Purpose:    This program is the driver to test the rational module.
//****************************************************************************
```

Declaration Comments

Variables must be declared one per line. Each variable can have a sidebar comment to the right indicating the variable’s purpose if the purpose of the variable is not totally obvious. Do not put any blank lines between the variables being declared. You must also group together variables that are related.

```
int seconds;
int minutes;
int hours ;
char firstName[MAXNAMESIZE];
char lastName[MAXNAMESIZE];
```

Function/Method Header

In the same way that a program header is used to describe the purpose of the program, the function/method header must be used to describe the purpose of the function. All your function/method headers must include the following:

- Method name
- Description
- Parameters
- Returned

```
//***************************************************************************
// Method:      setNumerator
//
// Description: Changes the value of the numerator to the value input.
//
// Parameters:  numerator   - numerator of the fraction
//
// Returned:    None
//***************************************************************************


void Rational::setNumerator (int numerator)
{
  this->numerator = numerator;
}
```

Sidebar and In-line Comments

A sidebar comment appears on the same line as the single statement it is describing. The comment must be brief and not exceed that line.

```
value <<= 1;   // multiply value by 2
```

In-line comments appear on their own lines and precede the segment of code they describe. You should use in-line comments to describe complex code that is not limited to a single statement. You should use blank lines to separate the comments from the segments of code they are describing. The comment below would not be placed in an actual program as it is obvious what the code is doing; however, the example illustrates what an in-line comment is to look like.

```
// Open input file for reading

ifstream inputFile;
inputFile.open(INPUT);
if(inputFile.fail())
{
  cout << “Could not open file”;
  exit(1);
}
```

Although using comments helps in describing your code, you must always make sure that your variables have meaningful names to make the code more understandable.

Printing

When printing your code you must use a fixed width font. Courier and Courier New are examples of fixed width fonts. You must also make sure that your lines do not wrap nor do they get cut off when printing. All printing is to be done in Portrait and the printing order for the files is as follows:

1) the program file containing main
2) class header (.h) / implementation (.cpp) pairs for each module

Note: Each module is to have a separate .h and .c file.

The final output you will turn in is to be printed in color since comments, keywords, strings, etc are highlighted for easy reading.