Coding Standards for C

Version 6.3

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 at Pacific University that use the C programming language. It is important for you to adhere to these standards in order to receive full credit on your assignments.

This 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 underscores such as MAX_NAME_CHARS. Further, your program is to contain no "magic constants." That is, all magic constants must be #defined to make program modification easier and program readabilty better. 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

```
input = fopen ("scores.dat", "r");
.....
for (indx = 0; indx < 100; indx++)
{
.....
}
```

Correct Program Style

```
#define MAX_SCORES 100
#define FILE_SCORES "scores.dat"

pScoresFile = fopen (FILE_SCORES, "r");
.....
for (indx = 0; indx < MAX_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 #defined.

Variable Names

1) A variable name is defined in all lowercase letters unless the variable name contains multiple names such as studentRecord. 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 col should be used.

3) Global variables must begin with g so that a name such as gHashTable denotes a global variable. Any variable that is global to only one file should be declared as ***static*** and use the gl (gee-el) prefix.

4) Funtions that are visible outside a module must have function names that begin with the name (or prefix) of the module in which they are found. Functions in a module that are not visible outside that module do not need to have the module name as part of the function name and the function must be declared as ***static***.

5) 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`
handle	`h`	`void **hWindow`
null terminated string	`sz`	`char szFileName`
structure	`s`	`Home sPerson`
function	`module name`	`stkPush`
globals	`g`	`char gNumFiles`
global to only one file	`gl`	`int glNumEntries`

Poor Program Style

```
int L (char *n)
{
  int i;

  for (i = 0; *(n + i) != '\0'; ++i)
  {
  }

  return i;
}
```

Good Program Style

```
int strLength (char *pszStr)
{
  int count;

  for (count = 0; '\0' != *(pszStr + count); ++count)
  {
  }

  return count;
}
```

Struct Names

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

Poor Program Style for Structs

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

Good Program Style for Structs

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

Implementation Example

The first file is a .h file that contains the definitions of the library (module). The second file is a .c (NOT .cpp) file that contains the actual implementation of the functions included in the library definition. The .c file includes the .h file at the top of the file.

Rational Example

```
/*************************************************************************
 File name:  rational.h
 Author:     Sharon Smith
 Date:       8/26/13
 Class:      CS300
 Assignment: Rational
 Purpose:    To define the header file for the rational module
*************************************************************************/

#ifndef RATIONAL_H
#define RATIONAL_H

typedef struct Rational
{
  int numerator;
  int denominator;
} Rational;

extern void       rationalPrint (Rational);
extern void       rationalSet (Rational *, int, int);
extern int        rationalIsEqual (Rational, Rational);
extern Rational   rationalMultiply (Rational, Rational);

#endif
```

```c
/*****************************************************************************
 File name:  rational.c
 Author:     Sharon Smith
 Date:       8/26/13
 Class:      CS300
 Assignment: Rational
 Purpose:    This is the implementation file for each of the rational
             functions associated with the rational module.
*****************************************************************************/


#include <stdio.h>
#include "../header/rational.h"

/*****************************************************************************
 Function:    rationalPrint

 Description: Outputs a fraction in the form
              numerator / denominator to the screen

 Parameters:  sRational - a fraction to be printed

 Returned:    None
*****************************************************************************/


void rationalPrint (Rational sRational)
{
  printf ("%d / %d", sRational.numerator, sRational.denominator);
}

/*****************************************************************************
 Function:    rationalSet

 Description: Initializes a fraction to the values of the numerator and
              denominator passed in.

 Parameters:  sRational   - a fraction
              numerator   - numerator initialization value
              denominator - denominator initialization value

 Returned:    None
*****************************************************************************/


void rationalSet (Rational *psRational, int numerator, int denominator)
{
  psRational->numerator = numerator;
  psRational->denominator = denominator;
}
```

```
/*************************************************************************
 Function:    rationalIsEqual

 Description: Compares two fractions returning a value of true if the
              numerators and denominators of both fractions are the same.

 Parameters:  sRational1 - first fraction used in comparison
              sRational2 - second fraction used in comparison

 Returned:    true if objects are equal; else, false
*************************************************************************/


int rationalIsEqual (Rational sRational1, Rational sRational2)
{
  return ((sRational1.numerator == sRational2.numerator) &&
          (sRational1.denominator == sRational2.denominator));
}

/*************************************************************************
 Function:    rationalMultiply

 Description: Multiples the numerators and denominators of two fractions.

 Parameters:  sRational1 - first rational number
              sRational2 - second rational number

 Returned:    A fraction that contains the result of the multiplication.
*************************************************************************/

Rational rationalMultiply (Rational sRational1, Rational sRational2)
{
  Rational sFraction;

  rationalSet (&sFraction, 0, 0);
  sFraction.numerator = sRational1.numerator * sRational2.numerator;
  sFraction.denominator = sRational1.denominator * sRational2.denominator;

  return sFraction;
}
```

Formatting

Indentation

Two spaces must be used as the unit of indentation per tab. Every IDE (Integrated Development Environment) such as Visual Studio or Eclipse includes an option for changing the number of spaces in a tab. These can usually be found in the preferences section.

Line Length

Lines must be no longer than 80 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 exactly 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 structs, functions) must appear on their own lines. Any code within the braces must be indented relative to the braces.

```
typedef struct Rational
{
  int numerator;
  int denominator;
} Rational;
```

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 module as briefly as possible. You must include the following sections in your module header:

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

```
/*****************************************************************************
 File name:  main.c
 Author:     Sharon Smith
 Date:       8/26/13
 Class:      CS300
 Assignment: Rational
 Purpose:    This program is the driver to test the rational module.
*****************************************************************************/
#include <stdio.h>
#include "../header/rational.h"

int main (int argc, char* argv[])
{
  Rational sRational1, sRational2;

  rationalSet (&sRational1, 1, 2);
  rationalSet (&sRational2, 2, 4);
  rationalPrint (sRational1);
  printf ("\n");
  rationalPrint (sRational2);
  printf ("\n");
  printf ("%i\n", rationalIsEqual ( sRational1,  sRational2));
  printf ("\n");
  rationalPrint (rationalMultiply (sRational1, sRational2));
  printf ("\n");
  return 0;
}
```

Declaration Comments

Variables must be declared so that comments are not necessary to explain the variable's meaning. You must also group together variables that are related.

```
int seconds;
int minutes;
int hours;
char *pszFirstName;
char *pszLastName;
```

Function Header

In the same way that a program header is used to describe the purpose of the program, the function header is used to describe the purpose of the function. Each function header must include the following:

- Function name
- Description
- Parameters
- Returned

Notice that each parameter exists on a single line and the – line up.

```
/*************************************************************************
 Function:    rationalSet

 Description: Initializes a fraction to the values of the numerator and
              denominator passed in.

 Parameters:  sRational   - a fraction
              numerator   - numerator initialization value
              denominator - denominator initialization value

 Returned:    None
*************************************************************************/


void rationalSet (Rational *psRational, int numerator, int denominator)
{
  psRational->numerator = numerator;
  psRational->denominator = denominator;
}
```

Function Pointers as Parameters

If a function pointer is provided as a parameter to a function, each parameter and the return value for the function pointer needs to be specified.

```
/*****************************************************************************
 Function:    htVisitAndUpdate

 Description: Visit each entry in the hash table and call the visitor function
              on each entry.  The visitor function takes a pointer to the
              entry (HT_DATATYPE*) so that visitor may update the data in
              the hash table.

              The parameters to the visitor function are:
               char*        - the key for the hash table entry
               unsigned int - the length of the key
               HT_DATATYPE* - the updatable entry in the hash table

              The visitor function returns void.

 Parameters:  psHashTable - the pointer to the hash table
              visitor     - the function pointer used to call a function on each
                            entry in the hash table

 Returned:  HT_ERROR_INVALID_HASHTABLE if the hash table in invalid
            HT_ERROR_NULL_PTR if the function pointer is NULL
            HT_NO_ERROR if no errors occur

*****************************************************************************/
ERRORCODE htVisitAndUpdate (HashTablePtr psHashTable,
            void(*visitor) (char*, unsigned int, HT_DATATYPE*))
```

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. Only document statements that are not obvious. The following documentation is not necessary for experienced C programmers but shows what a sidebar comment looks like.

```
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 must 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.

```
/* If the file exists, open the input file for reading */
```

```
if ((pInFile = fopen (FILE_SCORES, "r")) != NULL)
{
  ....
}
```

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) header (.h) / implementation (.c) 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.