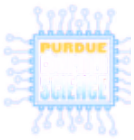
CS 240 - Programming in C

Spring 2025



C CODE STANDARD

I. NAMING CONVENTION

The following rules must be applied to all functions, structures, typedefs, unions, variables, etc.

A. Variable names should be in all lowercase.
If the name is composed of more than one word, then underscores must be used to separate them.

Example: int temperature = 0;

Example: int room_temperature = 0;

B. Use descriptive and meaningful names.

Example: Variable such as "room_temperature" is descriptive and meaningful, but "i" is not. An exception can be made if "i" is used for loop counting, array indexing, etc.

An exception can also be made if the variable name is something commonly used in a mathematical equation, and the code is implementing that equation.

C. All constants must be all uppercase, and contain at least two characters. Constants must be declared using #define.
A constant numeric value assigned must be enclosed in parenthesis.

String constants need to be placed in quotes but do not have surrounding parentheses.

Example: #define TEMPERATURE_OF_THE_ROOM (10)

Example: #define FILE_NAME "Data_File"

D. All global variables must be started with prefix "g_".
Declarations/definitions should be at the top of the file.

Example: int g_temperature = 0;

Global variable use should be avoided unless absolutely necessary.

II. LINE AND FUNCTION LENGTH

A. Each line must be kept within 80 columns in order to make sure the entire line will fit on printouts. If the line is too long, then it must be broken up into readable segments.
The indentation of the code on the following lines needs to be

at least 2 spaces.

Example: room_temperature = list_head->left_node->
                                    left_node->
                                    left_node->
                                    left_node->
                                    left_node->
                                    temperature;

Example: fread(&value, sizeof(double),
               1, special_fp);

B. Each function should be kept small for modularity purpose.
The suggested size is less than two pages.
Exception can be made, if the logic of the function requires its
size to be longer than two pages. Common sense needs to be followed.

Example: If a function contains more than two pages of printf
or switch statements, then it would be illogical to break
the function into smaller functions.

III. SPACING

A. One space must be placed after all structure control, and flow
commands. One space must also be present between the closing
parenthesis and opening brace.

Example: if (temperature == room_temperature) {

Example: while (temperature < room_temperature) {

B. One space must be placed before and after all logical, and
arithmetic operators; except for unary and data reference
operators (i.e. [], ., &, *, ->).

Example: temperature = room_temperature + offset;

Example: temperature = node->data;

Example: if (-temperature == room_temperature)

Example: for (i = 0; i < limit; ++i)

Example: *value = head->data;

C. One space must be placed after internal semi-colons and commas.

Example: for (i = 0; i < limit; ++i)

Example: printf("%f %f %f\n", temperature, volume, area);

D. #define expressions need to be grouped together and need
to be lined up in column 1. They need to have a blank line
above and below. Typically they should go at the top beneath
the includes.

Example: #include "hw1.h"

#define FUNCTION_NAME "Whatever"
#define UPPER_LIMIT (56)

```
            . . .

            /* whatever */
```

E. Never put trailing whitespace at the end of a line.

F. Never place spaces between function names and the parenthesis preceding the argument list.

IV. INDENTATION

A. Two space indentation must be applied to all structure, control, and flow commands. This two space indentation rule must be applied to the entire program.

Note that the opening brace must be placed on the same line as the structure, control, or flow command. The closing brace must be placed on the line after the structure, control, or flow commands. The closing brace must also be alone on the line. Even if only one statement is to be executed it is necessary to use braces.

Example:
```
for (i = 0; i <= size; ++i) {
  average += data[i];
}
```

Example:
```
while (temperature < MAX_TEMPERATURE) {
  average += data[i];
  temperature += offset;
}
```

Bad Example:
```
if (x < 7) {
  . . .
} else {  /* NO: else { should be on the next line */
  . . .
}
```

B. Parameters for functions with more than one parameter should be on the same line, unless the line length is exceeded. In that case, parameters on the next line should begin at the same column position as the parameters on the first line. The example below uses fewer than 80 characters just for demonstration purposes.

Example:
```
double average(double *data, int size, char *name,
               int temperature) {
  . . .
} /* average() */
```

Example:
```
int main(void) {
  . . .
} /* main() */
```

C. Do while loops need to be indented with the while on the same line as the closing brace.

Example:
```
do {
  . . .
} while (size < LIMIT);
```

V. COMMENTS

A. Comment should be meaningful and not repeat the obvious.
Comments are intended to alert people the intention of the code.

Example: This is a bad comment.

```
/* Variable to store the temperature */

double temperature = 0.0;
```

Example: This is a bad comment.

```
/* Increment i by one */

++i;
```

Example: This is a good comment.

```
/* Temperature is measured in Celsius */
/* and it ranges from 0 - 150 degrees */

double temperature = 0.0;
```

B. Place comments above the code rather than along side the code.
Exceptions can be made for short comments placed beside declarations,
else, and switch commands.

Example:
```
if (temperature == room_temperature) {
  statement;
  statement;
}
else {     /* comment for else statement */
  statement;
  statement;
}
```

Example:
```
switch (key) {
   case DO_THIS: {       /* comment for DO_THIS */
     statement;
     statement;
     break;
   }
   case DO_THAT: {       /* comment for DO_THAT */
     statement;
     statement;
     break;
   }
   default: {            /* comment for defaults */
     statement;
     statement;
     break;
   }
}
```

C. Code and comments must be separated by blank lines, and lined up.
A blank line is not required above the comment if it is the 1st.
line following an opening brace.

Example:
```
if (temperature == room_temperature) {
  statement;

  /* Comment */
```

```
            statement;
          }
          else {
            /* Comment for action when temperature is */
            /* not equal room_temperature             */

            statement;
          }
```

D. Function's name must be commented at the end of each function. The comment should be the name of the function indented one space after the closing brace and include left and right parentheses.

Example:
```
double average(double *data, int size) {
          . . .
} /* average() */
```

E. Function header comments should have a blank line above and below the comment.

See section VII for an example.

VI. MULTIPLE LOGICAL EXPRESSIONS

If multiple logical expressions are used, sub-expressions must be parenthesized. Note the spacing and format below.

Example:
```
if ((volume_box_a == volume_box_b) &&
    (volume_box_b == volume_box_c)) {
  . . .
}
```

Example
```
if (((side_a < side_b) && (time < max_time)) ||
    ((value  < data) && (limit > MIN_VALUE))) {
  . . .
}
```

VII. HEADERS

A header must be placed at the beginning of each function (including the main program). A header must contain detailed information, which describes the purpose of the function. The format is defined below: The header comment block must be at the left edge.

Example: Header at left edge.

```
/*
 *  This function computes the average of the passed data,
 *  which is stored in an array pointed to by ptr_data.
 *  The parameter Size is the index of the last array
 *  element used. The array uses items 0 to Size.
 */

double average(double *data, int size) {
  double average = 0.0;

  for (int i = 0; i <= size; ++i) {
    average += data[i];
  }
```

```
              return (average / (size + 1));

            } /* average() */
```

VIII. HEADER FILES

A. In general, every .c file should have an associated .h file.

B. Header files should be self-contained (compile on their own) and end in .h.

C. All header files should have #define guards to prevent multiple inclusions. The format of the symbol name should be _H.

   Example:

```
   #ifndef BAZ_H
   #define BAZ_H

   ...

   #endif // BAZ_H
```

D. All project header files should be listed as descendants of the project's source directory. UNIX directory shortcuts (e.g., . and ..) are forbidden.

   Example: src/base/logging.h should be included as:

```
   #include "base/logging.h"
```

E. Includes should be ordered per the following example. Suppose dir/foo.c implements things in dir2/foo.h. Your includes should be ordered as follows:

```
   #include "dir2/foo.h"

   #include <sys/types.h>
   #include <unistd.h>

   #include "base/basictypes.h"
   #include "base/commandflags.h"
   #include "foo/server/bar.h"
```

   Note the spaces. Any adjacent blank lines should be collapsed.

F. All relevant include files should be explicitly included. Do not assume that just because foo.h currently includes bar.h you do not need to explicitly include bar.h.

G. Non-local includes (standard libraries, etc) must be surrounded by < and >. Double quotes should only be used for include files that exist in the local directory structure.

IX. DEFENSIVE CODING TECHNIQUE

A. Return values of functions such as malloc, calloc, fopen, fread, fwrite, and system must be checked or returned whenever a possible error condition exists.

```
      Example: if ((data_fp = fopen(file_name, "r")) == NULL) {
                 fprintf(stderr, "Error: Cannot open file ");
                 fprintf(stderr, "%s.\n", file_name);
                 exit(TRUE);
               }
```

B. When the function fopen is invoked to open a file, fclose must be used to close the file. It is important to remember that fclose does not explicitly set the file pointer back to NULL. Therefore, it is necessary to set the file pointer to NULL.

C. After dynamically deallocating a pointer using free, the pointer must be set back to NULL.

D. Appropriate range checking must be performed to make sure received parameters are within expected range.

```
      Example: if ((temperature < LOWER_BOUND) ||
                   (temperature > HIGHER_BOUND)) {
                 fprintf(stderr, "Error: Temperature out of range.\n");
                 exit(TRUE);
               }

      Example:  assert((temperature > LOWER_BOUND) &&
                       (temperature <= HIGHER_BOUND));
```

E. The appropriately typed symbol should be used to represent 0 and NULL.

```
      Integers: Use 0
      Reals: Use 0.0
      Pointers: Use NULL
      Characters: Use '\0'
```

X. OUTPUT HANDLING

All error messages must be directed to standard error.

```
      Example: fprintf(stderr, "Error has occurred.\n");
```

XI. FORBIDDEN STATEMENTS

A. Do not use tabs for indentation.

B. Use only UNIX newline encoding (\n). DOS newlines (\r\n) are prohibited.

C. Do not make more than one assignment per expression.

```
      Bad Example: volume_box_a = volume_box_b = volume_box_c;

      Bad Example: if ((volume_box_a = compute_volume_box_a()) +
                       (volume_box_b = compute_volume_box_b()))
```

D. Do not use embedded constants; except for general initialization purposes and values that lack intrinsic meaning.
Common sense need to apply.

```
      Bad Example: return 1;                /* DO NOT USE */
                   In this case, the return value represents a failure.
                   Create a #defined constant at the beginning of your
```

```
                program instead and use that...
                #define HAILSTONE_ERROR  (1)

      Bad Example: if (temperature > 10)   /* DO NOT USE */
                Again, in this case the value means something, and you
                should define a constant instead...
                #define MAX_TEMPERATURE  (10)

      Bad Example: int temperature = 10;    /* DO NOT USE */
                Same idea...
                #define START_TEMP  (10)

      Bad Examples: #define ONE  (1)
                    #define FIRST_INDEX  (0)
                    #define FIRST_ITERATION  (0)

                If the value has intrinsic meaning, that meaning should
                be conveyed in the constant's name.

      Example: int temperature = 0;     /*  OK if 0 has no
                                         *  specific meaning
                                         */
     Some exceptions:

      Example: somefile_fp = fopen(file_name, "rb");

               The use of the embedded constant "rb" is ok.

      Example: if (fread(&data, sizeof(int), 2, somefile_fp) == 2)

               The embedded constants 2 are ok here.

      Example: if (fscanf(fp, "%d %s %f", &my_int, my_str, &my_float) == 3)

               The embedded constant 3 is okay here.

      Example: some_string[MAX_SIZE - 1] = '\0';

               Subtracting or adding one from/to the size for the NULL
               terminator is fine.

      Example: if (strchr(some_buf, 's')) {

               You do not need to define a constant when referring to
               single, human-readable characters.

      Example: malloc(sizeof(something) * 4);  /* Allocate space for 4 somethin

               The constant 4 here is okay.

      Again, please use common sense.

   E. The use of goto is forbidden in this course.

XII. VARIABLE DEFINITIONS

    A. No more than one variable may be defined on a single line.

       DON'T DO THIS:

       int side_a, side_b, side_c = 0;
```

Do it this way:

```
int side_a = 0;
int side_b = 0;
int side_c = 0;
```

B. All variables must be initialized at the time they are defined.

C. Variables should be placed in as local a scope as possible, as close to the first use as possible.

Example:

```
while (const char *p = strchr(str, '/')) {
  str = p + 1;
}
```

D. Variable length arrays are prohibited in this course.