Updated: 10-5-2013

- **Programming Standards - A Style Guide for Writing C Programs**

  - **Commenting, Readability, Clarity, etc:**
    - Why worry about programming style? Who cares what a program reads like if it works? Does it not take too much of my precious time to make it look pretty? Are not the rules arbitrary, anyways?
    - The answer is easy. Well-written code is easy to read and understand, almost always has fewer errors, and is more efficient than code that has been carelessly mixed together and never polished.
    - In the rush to get programs out the door to meet assignment or project deadlines, it is easy to push style aside and worry about it later. This can be a costly decision because what is being generated is in most cases, sloppy code. Sloppy code is bad code – awkward and hard to read, and often buggy.
    - Good programming style must be a matter of habit. Think about style as you develop code originally. Take the time to read and debug your code. Invest time to revise and improve your code. Do the above and you are developing good habits. Once they become a habit, they enter your subconscious and you will always write good code even under the worst of constraints such as time, pressure from your boss (or teacher) to complete your tasks NOW, or even in an interview.
    - The following is a list of limited guidelines. Some come from the experience of having developed code, some from making horribly costly mistakes under serious time constraints, some from supervising a team of programmers and learning new things from them, while at the same correcting mistakes, and some have come from reading the following books:
      - The C Programming Language, Kernighan and Ritchie.
      - The C++ Programming Language, Bjarne Stroustroup.
      - Writing Solid Code, Steve Maguire.
      - Code Complete, A Practical Handbook of Software Construction, Steve McConnell.
      - The Practice of Programming, Kernighan and Pike.
      - The Art of Computer Programming (three volumes), Donald Knuth.
      - Programming Pearls, Jon Bentley.

  - **Naming Conventions:**
    - Naming conventions specify the rules used to make up a name in your program. A variable or function name labels an object and conveys information about its purpose.
    - Naming conventions make it easier for yourselves and me to understand your code. It also makes it easier to understand and reuse the code for an assignment later in the semester.
    - There are many naming conventions. Common ones include using names that begin or end with `p` for pointers, as in, `queuep`; initial capital letters for `Globals`; all capital letters for `CONSTANTS`; initial capital letters for function names, as in, `GetName`. More sweeping rules would encode the type and usage information in the variable names as in `pd` to mean "pointer to `double`"; `iCnt` to mean an integer variable that keeps a running count; `psSrc` to mean a pointer to an array of characters that will be read from.
    - Specific rules are not very important and there will be no imposition of any naming conventions for CS120 assignments.
    - The important thing is to find a simple and sensible convention and to follow it religiously. It is one of the factors that will be checked for in your hardcopy to determine your grade.
    - Here are some useful guidelines.
      - Create a name for a variable or function that is informative, concise, memorable, and if possible pronounceable.
      - The amount of information that needs to be gleaned from the name comes from context and scope.
      - Global variables are visible to many programmers. They require more descriptive names and comments in contrast to local names. For example, if we have a global variable that keeps track of pending users in a queue, the declaration might be:

```
/* current length of input queue */
int gPendingUsers = 0; /* ok */
int gPending = 0;      /* ok */
int n = 0;             /* not ok */
```

- Shorter names suffice for local variables. For example, an automatic variable keeping track of the number of vertices of an 3d object, might be declared as:

```
int points; /* ok */
int num_points; /* ok */
int numberOfPoints; /* probably too much */
```

- Local variables used in conventional ways can have very short names. The use of `i` and `j` for loop indices, `p` and `q` for pointers, and `s` and `t` for strings is so frequent that there is little to be gained and perhaps some loss in longer names. Compare:

```
for (theArrayIndex = 0; theArrayIndex < numberOfElements; theArrayIndex++)
  elementArray[theArrayIndex] = elementArray;
```

to

```
for (i = 0; i < nsize; i++)
   elem[i] = i;
```

- Important thing to remember: clarity is often achieved through brevity.

- Name consistently
  - Related things must be given related names that exhibit their relationship and highlight their differences.
  - Consider the following `struct` and function declarations:

```
typedef struct UserStack_
{
  int noOfItemsS;
  int topOfTheStack;
  int stackCapacity;
}UserStack;

int NoOfUsersInStack(UserStack *pus) { … }
```

In addition to being too long, the member names are highly inconsistent. For example, the word "`stack`" appears as `S`, `Stack`, and `stack`.

  - Think of an alternative naming strategy. Since all of the named members are accessed through an object of type `UserStack`, there is no necessity for member names to mention, "`stack`" at all. That is,

```
Stack.stackCapacity
```

is highly redundant. Remember, in a lot of situation context suffices.

  - This version reads and writes better

```
typedef struct UserStack_
{
  int num_items;
  int top;
  int capacity;
};

int GetNumUsers(UserStack *pus) { … }
```

since it leads to code like

```
UserStack stack;
…
stack.capacity++;
n = GetNumUsers(&stack);
```

- Function names
  - Function names should be based on active verbs, perhaps followed by nouns.

```
struct Date today;
today = GetTime(&today);
```

- Functions that return a Boolean result should be so named so that the return value is unambiguous. For example,

```
if (CheckLeapYear(year))
    …
```

does not indicate whether `true` means `year` is a leap year or if `false` indicates that `year` is a leap year. On the other hand,

```
if (IsLeapYear(year))
 …
```

makes it clear that the function returns `true` if `year` is is a leap year, otherwise `false`.

- A name not only labels, it conveys information to the reader. A misleading name can result in mystifying bugs. Consider the following function:

```
int WithinBounds(char *ps, char ch)
{
     /* returns a value between 0 and strlen(ps)-1 if it finds an */
     /* occurrence of ch in string ps, else it returns strlen(ps) */
   int j = GetIndex(ps, ch);
   int n = strlen(ps);

   return (j == n);
}
```

In the above function, `GetIndex()` returns a value between `0` and `strlen(ps)-1` if it finds an occurrence of `ch` in string `ps`, otherwise it returns `strlen(ps)`. The `bool` value returned by `WithinBounds()` is thus the opposite of what the name implies. At the time this code was written, the behavior is well understood by the original programmer. If the program is modified later, by a different programmer, the name is sure to confuse because the new programmer would not know if the behavior is a bug or if the function was named incorrectly.

- **Expressions and Statements**
  - Write expressions and statements in a way that makes their meaning as transparent as possible. That is, write the clearest code that does the job.
  - Format to help readability. For example, use spaces around operators to suggest grouping and precedence.
  - Indent to show structure.
    - A consistent indentation style is the lowest-energy way to make a program's structure self-evident. An example of a badly formatted is:

```
for(n++;n<100;field[n++]='\0');
*i = '\0'; return ('\n');
```

    - Reformatting improves it a little bit:

```
for (n++; n < 100; field[n++] = '\0')
   ;
*i = '\0';
return('\n');
```

    - Even better is to put the assignment in the body and separate the increment, so the loop takes a more conventional form and is thus easier to understand for both the reader and the programmer:

```
for (n++; n < 100; n++)
   field[n] = '\0';

*i = '\0';
return ('\n');
```

  - Use the natural form of expressions.
    - Write expressions as you might speak them aloud.
    - Conditional expressions that include negations are always hard to understand. Removing the negation makes the code read more naturally. Consider the following expression:

```
if (!(force < minForce) && !(force >= maxForce))
    …
```

Each test is stated with a negation, though there is no need for either to be. Changing the relations around lets us state the tests positively without a negation:

```
if ((force >= minForce) && (force < maxForce))
    …
```

- Parenthesize to resolve ambiguity.
  - Parentheses specify grouping and can be used to make the programmer's intent clear even when they are not required. The inner parentheses in the previous example are not required but they improve the readability of the expression.
  - C has lots of nasty precedence problems and it is easy, even for a seasoned programmer, to make a mistake.
  - When mixing unrelated operators, it is never a bad thing to parenthesize your expression. Since logical operators have greater precedence than assignment, parentheses are required for most expressions that combine them:

```
while ((ch = getchar()) != EOF)
    …
```

The bitwise operators `&` and `|` have lower precedence than relational operators like ==. For example, the expression:

```
if (flag&MASK == BITS)
    …
```

actually means

```
if (flag & (MASK==BITS))
    …
```

which was probably not the programmer's intent. Because it combines bitwise and relational operators, the expression needs parentheses:

```
if ((flag&MASK) == BITS)
    …
```

  - Even if parentheses are not necessary, they can aid if the grouping is hard to understand at first glance. The following expression does not need parentheses:

```
leapYear = year % 4 == 0 && year % 100 != 0 || y % 400 == 0;
```

but the use of parentheses makes it easier to understand the structure of the expression:

```
leapYear = ((year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0));
```

- Break up complex expressions.
  - C contains a rich set of operators and expression syntax. It is easy to get carried away and load everything into a single statement.
  - The following statement is compact but it contains too many operations into a single statement. These kinds of constructions are undesirable because they require a considerable amount of time to write and greater amounts of time to understand.

```
*pi += (*pi2=(5*k < (m-n) ? ai[k+1] : ai2[k--]));
```

It is easier to understand when divided into several statements:

```
if (5 * k < m - n)
   *pi2 = ai[k + 1];
else
   *pi = ai2[k--];

*pi += *pi2;
```

- Be clear
  - Spend your creative energy in writing clear code not clever code.
  - Consider the following intricate expression:

```
result = result >> (flag – ((flag >> 3) << 3));
```

    The innermost expression shifts `flag` by 3 bits to the right and the result is shifted to the left, thereby clearing the least-significant three bits of `flag` to 0. This result in turn is subtracted from the original value, giving the least-significant three bits of `flag`. The value of these three bits is used to shift `result` to the right.
  - That is, the original expression is equivalent to the following expression:

```
result = result >> (flag & 0x07);
```

    which can be better rewritten as:

```
result >>= flag & 0x07;
```

    It takes considerably more time to puzzle out what the first version is doing, while the second and third versions are far easier to read and understand.
  - Another example uses the `?:` operator:

```
child = (!LC && !RC) ? 0 : (!LC ? RC : LC);
```

    To figure out the meaning of the above construct, the reader needs to follow all possible paths through the expression. The following form is longer, but easier to understand because it makes the paths explicit.

```
if (LC == 0 && RC == 0)
   child = 0;
else if (LC == 0)
   child = RC;
else
  child = LC;
```

  - The important thing to remember is that clarity is not the same as brevity. The proper criterion to be used when writing code is understandability (if there is such a word). In the first example the clearer code is shorter, while in the second example it is longer.
  - Another import thing to realize, that many new programmers miss, is that shorter statements are not always faster to execute. In most situations described above, all of the statements take about the same time to execute, regardless of the formatting or total number of statements.

- **Consistency**
  - Consistency leads to better programs. In inconsistent programs:
    - Formatting varies unpredictably
    - Loop over an array runs from the first element to the last on some occasions and from the last element to the first other times.
    - Strings are copied with the `strcpy()` function in some situations, but other times with `for` loop.
  - Such variations make it harder to understand the program. But if the same computation is done the same way every time it appears, any variations suggest a genuine difference – one worth noting.
  - Use a consistent indentation and brace style.
    - Proper indentation brings out the structure of the program.
    - Which style to use? Should the opening brace be on the next line as the `if` or follow on the same line? Arguments over which style is the best are a waste of time. Pick one specific style and use it consistently.
    - Are braces to be included even when they are not needed? Like parentheses, braces can resolve ambiguity and help in generating cleaner code. For consistency, most experienced programmers always put braces around loop or `if` bodies. If the body contains a single statement, some tend to omit them.
    - In some situations such as the "dangling `else`" problem, braces may be required. Consider the following code that contains a logic error that is disguised by indentation.

```
if (month == FEBRUARY)
{
   if (LeapYear(year))
     if (day > 29)
       legal = FALSE;
    else
      if (day > 28)
        legal = FALSE;
}
```

The indentation is misleading because the `else` is attached to the

```
if (day > 29)
```

expression, thereby leading to incorrect code.

- Always use braces when one `if` immediately follows another.

```
if (month == FEBRUARY)
{
   if (LeapYear(year))
  {
    if (day > 29)
       legal = FALSE;
  }
  else
  {
   if (day > 28)
     legal = FALSE;
  }
}
```

- Even with the proper use of braces, the program is still a little bit hard to read because of the presence of quite a few `if` expressions. The use of an extra variable makes the code easier to understand.

```
if (month == FEBRUARY)
{
   int nday = (LeapYear(year)) ? 29 : 28;
   legal = (day > nday) ? FALSE : legal;
}
```

- Use idioms for consistency.
  - Natural languages contain idioms that represent conventional ways in which the words of the language are combined to express ideas. Native speakers of a particular language express certain ideas in the same idiomatic fashion.
  - In the same manner, most experienced C/C++ programmers express certain ideas in the same common way. An example of a common idiom is the `for` loop. Here are different ways of writing the same `for` loop. The first way to write the loop is:

```
i = 0;
while (i <= n - 1)
   ai[i++] = 0;
```

Some people might write the loop as:

```
for (i = 0; i < n; )
   ai[i++] = 0;
```

or even as:

```
for (i = n; --i >= 0; )
   ai[i++] = 0;
```

All of the above methods are correct, but the idiomatic form is:

```
for (i = 0; i < n; ++i)
   ai[i] = 0;
```

The idiomatic form visits each member of a `n`-element array indexed from `0` to `n-1`. It places the loop control in the `for` itself, runs in increasing order, uses the idiomatic `++` operator to update the index, and it leaves the index at a known value one past the last array element. Native speakers of C and C++ recognize the idiomatic form without study and write it correctly without a moment's thought.

- For an infinite loop, the idiomatic form is

```
for (;;)
    …
```

or

```
while (1)
…
```

However, the second form may cause a compiler warning.

- Consistent use of idioms draws attention to non-standard loops.

```
int i, *pi, size;
…
/* make sure to check if the operator new allocated  */
/* size * sizeof(int) number of bytes from the heap. */
pi = new int[size];
for (i = 0; i <= size; i++)
   pi[i] = i;
```

Since the loop test is <= the loop continues after the end of the array overwriting whatever is stored next in memory.

- Indentation should be idiomatic too. The vertical layout in the following `for` loop detracts from readability; it looks like three statements, not a loop. Another factor to consider is that the sprawled layout might force code onto multiple screens, thereby detracting from readability.

```
for (
   pi = ai;
   pi < ai + 128;
   *pi++
   )
{
    ;
}
```

A standard loop is much easier to read:

```
for (pi = ai; pi < ai + 128; pi++)
   *pi = 0;
```

- Another common idiom is to nest an assignment inside a loop condition, as in:

```
while ((ch = getchar()) != EOF)
   putchar(ch);
```

- C/C++ have idioms for allocating space for strings and manipulating them because of the null terminating character.

```
char *pc, buff[256] = "Hello World!";
…
pc = new char[strlen(buf) + 1];
…
```

Beware if you don't see the +1 when allocating memory for strings.

- Use `else-if`s for multi-way decisions.
  - A sequence of nested `if` statements is an indication of awkward code which might contain errors.

```
if (argc == 3)
   if ((pfin = fopen(argv[1], "r")) != NULL)
     if ((pfout = fopen(argv[2], "w")) != NULL)
     {
       while ((ch = getc(pfin)) != EOF)
         putc(ch, pfout);
       fclose(pfin);
       fclose(pfout);
     }
     else
       printf("Cannot create output file %s\n", argv[2]);
     else
       printf("Cannot open input file %s\n", argv[1]);
else
   printf("Usage: cp.exe input_file output_file\n");
```

  The sequence of `if`s requires us to maintain a mental stack of what tests were made, so that at the appropriate point, if we can still remember, we can pop them until we determine the corresponding action. Since at most one action is required, use an `else if` statement.
  - Multiway decisions are idiomatically written as a chain of `if … else if … else` statements:

```
if (condition 1)
   statement 1
else if (condition 2)
   statement 2
…
else if (condition n)
   statement n
else
   default statement
```

  Make sure to align all of the `else` clauses vertically rather than lining up each `else` with the corresponding `if`. Vertical alignment emphasizes that the tests are done in sequence and keeps them from marching off the right side of the page.
  - We write the original using a cleaner format and clearer logic that corrects the problem in the original.

```
if (argc != 3)
  printf("Usage: cp.exe input_file output_file\n");
else if ((pfin = fopen(argv[1], "r")) == NULL)
  printf("Cannot open input file %s\n", argv[1]);
else if ((pfout = fopen(argv[2], "w")) != NULL)
{
  printf("Cannot create output file %s\n", argv[2]);
  close(pfin);
}
else
{
  while ((ch = getc(pfin)) != EOF)
    putc(ch, pfout);

  fclose(pfin);
  fclose(pfout);
}
```

  The rule is to follow each decision as closely as possible by its associated action. That is each time a test is made, something is done.
  - Attempts to re-use small pieces of code lead to program structures that are not idiomatic.

```
switch(ch)
{
  case '-': sign = -1;
  case '+': ch = getchar();
  case '.': break;
  default: if (!isdigit(ch)) return 0;
}
```

The use of a tricky sequence of fall-through in a `switch` statement to avoid duplicating one line of code is not idiomatic. The cases in a `switch` statement almost always must end with a `break`, the rare exceptions clearly commented. A more idiomatic form that is longer but easier to read is:

```
switch(ch)
{
  case '-':
    sign = -1;
    /* fall through */
  case '+':
    ch = getchar();
    break;
  case '.':
    break;
  default:
    if (!isidigit(ch))
      return 0;
    break;
}
```

Also note that for such an unusual structure a sequence of `else-if` statements is even clearer:

```
if (ch == '-')
{
  sign = -1;
  ch = getchar();
}
else if (ch == '+')
{
  ch = getchar();
}
else if (ch != '.' && !isdigit(ch))
{
  return 0;
}
```

When several cases have identical code, the use of fall-through is acceptable. The idiomatic layout of such a `switch` statement is:

```
case '0':
case '1':
case '2':
   …
  break;
```

- **Function Macros**
  - C programmers sometimes tend to write macros instead of functions for very short computations that will be executed frequently. Good examples are finding the absolute value of an integer, finding the minimum or maximum values given two values, and I/O operations. The most often quoted reason for the use of macros is performance since they tend to avoid the overhead of a function call. The reason might have been valid when C was first defined around thirty years back. With modern processors and compilers, there is never a good reason for the use of macros in place of functions.
  - Macros are hard to write. In most cases, they create more insidious problems than the performance overhead that they overcome.
  - The biggest problem with function macros is that a parameter that appears more than once in the definition might be evaluated more than once. Consider the following macro:

```
#define ISUPPER(ch) ((ch) >= 'A' && (ch) <= 'Z')
```

If the macro is called in a context like this:

```
while (ISUPPER(ch = getchar()))
   …
```

then each time an input character is greater than or equal to `A`, it will be discarded and another character read to be tested again `Z`.

- Consider the following situation where the use of a macro increases performance overhead rather than decreasing it.

```
#define ROUND_TO_INT(x) ((int) ((x)+(((x)>0.)?0.5:-0.5)))

…

distance = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

  The square root computation is performed twice as often as necessary.
- In C++, inline functions make function macros obsolete.

- **Magic Numbers**
  - Magic numbers are literal numeric values that make their appearance in programs.
  - Give names to magic numbers.
    - A literal value in source code provides no indication of its derivation or importance, making the code harder to read, understand, and modify. Consider the following excerpt from a program to print a histogram of letter frequencies on a 24 by 80 cursor-addressed terminal. The program is unnecessarily opaque because of the use of a large number of magic numbers.

```
/* compute scale factor */
scale = limit / 20;
scale = (scale < 1) ? 1 : scale;

/* generate histogram */
for (i = 0, column = 0; i < 27; i++, j++)
{
   column += 3;
   k = 21 - (letter[i] / scale);
   star = (letter[i] == 0) ? ' ' : '*';
   for (j = k; j < 22; j++)
     Draw(j, column, star);
}

/* label X axis */
Draw(23, 2, ' ');
for (i = 'A'; i <= 'Z'; i++)
   printf("%c ", i);
```

      The code includes the numbers 20, 21, 22, 23, and 27. There is no way to determine if there is a relation between numbers or not. Also, since the program intent is to print a histogram on a terminal, there are 3 important values: 24, the number of rows; 80, the number of columns; and 26, the number of letters in the alphabet. But none of these numbers make an appearance, increasing the complexity of the code.

      By giving names to numbers in the program, we can write clearer and cleaner code.

```
enum
{
  MINROW = 1,   /* top edge            */
  MINCOL = 1,   /* left edge           */
  MAXROW = 24,  /* bottom edge (<=)    */
  MAXCOL = 80,  /* right edge (<=)     */
  LABELROW = 1, /* position of labels  */
  NUMLET = 26,  /* number of alphabets */
  HEIGHT = MAXROW - 4,         /* height of bars */
  WIDTH  = (MAXCOL - 1)/NUMLET /* width of bars  */
};

…

/* compute scale factor */
scale = (limit + HEIGHT - 1) / HEIGHT;
scale = (scale < 1) ? 1 : scale;

/* generate histogram */
for (i = 0; i < NUMLET; i++)
{
  if (letter[i] == 0)
```

```
  continue;
  for (j = HEIGHT - letter[i] / scale; j < HEIGHT; j++)
  Draw(j + 1 + LABELROW, (I + 1) * WIDTH, '*');
}

/* label X axis */
Draw(MAXROW - 1, MINCOL + 1, ' ');
for (i = 'A'; i <= 'Z'; i++)
  printf("%c ", i);
```

- Define numbers as constants, not macros.
  - C programmers have traditionally used the preprocessor directive `#define` to manage magic number values. Both C and C++ preprocessors are powerful but blunt tools to manage magic numbers because they change the lexical structure of the program.
  - On the other hand, both C and C++ contain facilities that allow for constants to be defined. Integer constants can be defied with an `enum` declaration, as in the previous example. Constants of any type can be declared with `const`:

```
const int MAXROW = 24, MAXCOL = 80;
int ai[MAXROW]; /* ok in C++, error in C */
```

- When testing characters, use character constants, not integers.
  - A test like this:

```
if (ch >= 65 && ch <= 90)
   …
```

  is harder to read and understand, as compared to:

```
if (ch >= 'A' && ch <= 'Z')
   …
```

- `0`
  - The number `0` appears often in C and C++ programs, in many contexts. The compiler will convert the number into the appropriate type, but it helps the reader to understand of each `0` if the type is explicit. For example:

```
str = 0;
name[j] = 0;
d = 0;
```

  must be re-written as:

```
str = NULL; /* in C */
str = 0;    /* In C++, 0 is the accepted convention for null pointer */
name[j] = '\0';
d = 0.0;
```

- Use the language to calculate the size of an object.
  - Do not use an explicit size for any data type; use the `sizeof` operator.
  - The operand of the `sizeof` operator is evaluated at compile-time, so there are no performance overheads to its use. Therefore, use `sizeof(int)` rather an explicit value such as `2` or `4`.
  - It is definitely worth it to write code that does not have change if the type or size of the array changes. For example, use `sizeof(array[0])` rather than `sizeof(int)` because it is one less to change if the type of array changes from `int`, to say, `double`. The second example shows how the code does not change even if the size of the array changes:

```
void foo(int *, int size)
{
   …
}

int main()
{
   int array[100];
```

```
    foo(array, sizeof(array) / sizeof(int));

    …
}
```

- **Comments**
  - Comments are used to aid the reader of a program. They do not help by stating things the code obviously says, or by contradicting the code, or by distracting the reader with elaborate typographical displays.
  - The best comments aid the understanding of a program by briefly pointing out the salient details and by providing a larger-scale view of the proceedings.
  - Comments are used to add something that is not immediately evident from the code, or collect in one place information that is spread throughout the source.
  - Do not belabor the obvious.
    - Comments should not report self-evident information, such as `i++` has incremented `i` by 1.
    - Such comments detract from the readability of the program by adding clutter.
    - Some examples of unnecessary comments:

```
/* default */
default:
  break;

/* return FAILURE */
return FAILURE;

/* set pointer to point to nothing */
pi = 0;

/* increment queue counter by 1 */
queuecount++;

/* assign "total" to "capacity" */
pStack->total = pStack->capacity;
```

  - Comment functions and global data.
    - Students are taught that it is important to comment all their code. Similar is the case with professional programmers. The purpose of commenting might be lost in following these rules blindly.
    - Comment functions, global variables, constant definitions, fields in structures, union, and classes, and anything else where a brief summary might aid in understanding. The aim is to clarify, not confuse.
    - Since global variables appear at different places in a program and are visible to other programmers, a comment is always required so that other can refer to it when required.
    - All functions must be commented briefly to bring out its salient features. In addition, every function must include comments that provide, in pseudo-code, the high-level implementation. Here is an example for a function `StrLen()` that returns the length of a null-terminated string.

```
/* Start Header -------------------------------------------------------------
  DESCRIPTION: Return the number of characters in null-terminated
               string str, not including the null terminating character.

  METHOD:      for each character of string str[] do:
               if str[i] is equivalent to null-terminating character -
                  we are done - and we return i
- End Header -------------------------------------------------------------*/
int StrLen(char *str)
{
   …
}
```

    - Do not comment bad code. Rewrite it. Good code needs fewer comments than bad code. Comment anything unusual or confusing, but when the comment outweighs the code, the code probably needs to be rewritten. Consider the following code:

```
/* If "result" is 0 a match was found so return true (non-zero).
 * Otherwise, "result" is non-zero so return false (zero).
 */
printf("Returns !result = %d\n", !result);
return(!result);
```

Negations, remember, are hard to understand and are to be avoided. Part of the problem is the uninformative variable name, `result`. A more descriptive name, `matchfound`, makes the comment unnecessary and cleans up the print statement, too:

```
printf("Returns matchfound = %d\n", matchfound);
return(matchfound);
```

- Do not contradict the code. Most of the comments agree with the code when they are written, but as bugs are fixed and the program evolves, the comments are often left in their original form, resulting in disagreement with the code. When a comment contradicts the code, a lot of time might be spent in unnecessarily debugging the code, because the comment was taken as truth. When code evolves, comments should too.

- **Additional Standards for C Programs in CS120**

  - Local Variable Declarations/Definitions- All variable declarations need to have a comment to the right that explains the purpose of the variable. There are no exceptions to this rule. No single-letter variable names are allowed. The only exceptions to the "single-letter" rule are variables used as indexes in a for loop. Only one variable per line may be declared. The exception to the "one variable per line" rule is if you have several loop counters and you want to declare them at the same time. If the comment is too long and won't fit to the right, put it on the line above the variable. All comments that are to the right need to be aligned and should be close to the longest variable declaration as such:

```
int i, j, k;  /* for loop counters, single-letter OK here */
int tabs = 0; /* Number of tabs in file                   */
int length;   /* Length of string                         */
int begin;    /* The left end of the string               */
int end;      /* The right end of the string              */
```

  - File Header Comments – All files must contain a comment at the top with information such as the student's name, the section, due date, etc., all aligned properly. This is an example of a file header comment:

```
/*****************************************************************************
filename    marathon.c
author      Nigel Tufnel
DP email    tap@digipen.edu
course      CS120
section     D
assignment  9
due date    1/21/2018

Brief Description:
  This program contains the calculation to convert the distance of a
  marathon in miles to kilometers.

*****************************************************************************/
```

- Function Header Comments – All functions must include a function header comment immediately above the function. There are 4 parts to the comment: The function name, a description, inputs, and outputs. The fields must be named as shown and must line up exactly as shown. Here is an example:

```
/*****************************************************************************
   Function: mystrlen

Description: Counts the number of characters in a string. Doesn't include
             the terminating NULL character.

     Inputs: text - A NULL-terminated C-style string.

    Outputs: The length of the string (integer), not including the
             NULL terminator.
*****************************************************************************/
int mystrlen(const char *text)
{
  /* Function body goes here... */
}
```

  There should be no blank lines between the header comment and the function itself. However, there should be at least one blank line between the closing brace of the function body and the following function header comment (which is for the function that follows).

- Functions declarations (and definitions) can not be over 80 characters in length, so you will have to format them manually. The function declaration for `jumble` is over 100 characters, so you need to put it on multiple lines. Notice how the parameters are all left aligned right after the left parenthesis. This formatting also allows you to put a comment next to each parameter if you need to describe them:

```
void jumble(unsigned char *string,          /* a comment could go here */
            const unsigned char *password, /* a comment could go here */
            enum CODE_METHOD method,       /* a comment could go here */
            int passes                     /* a comment could go here */
           );
```

- Most functions that return a value will have 3 sections: declarations, executable statements, return statement. There should be a blank line between these sections. Some functions will have several blocks of executable statements and they should also be separated by a blank line.

Example formatting for a simple function

```
int foo(int param1, int param2)
{
  int var1;     /* comment */
  int var2 = 1; /* comment */                 <==== Declarations
  int var3 = 2; /* comment */

  for (var1 = 0; < param1; var1++)
  {
    var2 += param2;          /* necessary */
    var3 -= var2 - param2;   /* comments  */   <==== Executable statements
    if (var3 == 0)           /* can go    */
      var2 = 0;              /* here      */
  }

  return var1 * var2; /* comment */           <==== return statement
}
```

- Internal Documentation (comments). All code needs to have some documentation associated with it. The amount of detail in the comments depends on the code. Typically, more complex code requires more detailed documentation. Comments can either be placed to the right of the statements they are describing, or on a line immediately above it. Don't put comments to the right unless the comment will fit on one line. Remember, your comments should be explaining the purpose of the code, not re-stating the obvious. Sometimes even a simple statement can be more easily understood by the programmer if it includes a useful comment. The first comment below is useless. The second comment says something that will help a programmer reading the code:

```
pCursor++; /* Increment pCursor (useless comment)     */
pCursor++; /* Point to the next line in the paragraph */
```

  You can put the comment above the statement instead of to the right, but never below the statement:

```
  /* Increment pCursor (useless comment)     */
pCursor++;

  /* Point to the next line in the paragraph */
pCursor++;
```

- Multi-line comments – When using multiple lines for comments, you should format them as such:

```
/*
 * This is a long comment because the line of code below is too
 * complicated to describe in one line. I should really clean up
 * the code so I don't have to write a book about it. But, I don't
 * have enough time to write short code with short comments, so I'm writing
 * this massive missive instead.
 */
```

  Multi-line comments should go above the statement(s) they describe rather than to the right.

- All `#include` directives must have a comment next to them listing the reason for the included files:

```
#include <stdio.h>  /* printf, scanf, fopen */
#include <stdlib.h> /* rand, atoi, srand    */
#include <time.h>   /* time, clock          */
```

- No global variables are permitted in any code. The only global objects are functions, enumerations, constants, and `#defines`.

- No lines longer than 80 characters.

- No tabs. Use spaces instead. Most text editors allow you to configure them so that, when you press the tab key, spaces are inserted instead of the tab character. The number of spaces to use for a tab is 2.

- Indentation – Proper indentation is a must for readability. In this class, the proper size for indentation (tabs) is 2 spaces.

- Braces – Curly braces should always be on a line by themselves and should be aligned vertically.

Proper formatting for a 'for' statement

```
for (i = 0; i < 10; i++)
  a[i] = 0;

for (i = 0; i < 10; i++)
{
  if (calc(i) < 0)
    break;

  a[i] = calc(i * i);
  b[i] = calc(a[i]);
}
```

Proper formatting of a 'while' statement:

```
while (i < 10)
  a[i++] = 0;

while (i < 10)
{
  a[i] = 0;
  i++;
}
```

Proper formatting for an 'if..else if' statement

```
if (year == 1)
  printf("Freshman\n");
else if (year == 2)
  printf("Sophomore\n");
else if (year == 3)
  printf("Junior\n");
else if (year == 4)
  printf("Senior\n");
else
{
  printf("Invalid year\n");
  printf("Please re-enter it.\n");
}
```

Proper formatting of a 'switch' statement:

```
switch (year)
{
  case 1:
    printf("Freshman\n");
    break;
  case 2:
    printf("Sophomore\n");
    break;
  case 3:
    printf("Junior\n");
    break;
  case 4:
    printf("Senior\n");
    break;
  default:
    printf("Invalid year\n");
    break;
}
```

Proper formatting for a 'do...while' statement

```
do
{
  printf("Enter a number: ");
  scanf("%d", &number);

  printf("You entered %i\n", number);

  printf("Enter another number? (1=yes,0=no) ");
  scanf("%d", &choice);
}
while (choice != 0);
```

- `#defines` – Defines for the preprocessor should be used sparingly and should only be used to introduce a constant into the program. All defines must be in all uppercase:

```
#define YARDS_PER_MILE 1760F
#define KILOS_PER_MILE 1.609
#define PI 3.1415926535
```

  C++ note – The const keyword behaves differently in C than in C++, so often you must use a `#define` in C where a `const` would be used in C++.

- Operators and Spacing – Spaces must surround all binary operators. Unary operators should not have any extra spaces:

```
r = (w + x) * (y + z);
a = -b + c;
f = sqrt(b * b – 4 * a * c);
w = --x + (y++ / z);
theta = sin(a) * -cos(b);
```

- Function declarations (prototypes) – Although function prototypes do not require names for their parameters, it's good practice to do so. In CS120, all function declarations must include meaningful names for the parameters.

```
  /* Acceptable prototype */
int Calculate(float rate, float time, float distance);

  /* Unacceptable prototypes */
int Calculate(float r, float t, float d);
int Calculate(float, float, float);
```

- Calling functions – No space between the function name and left parenthesis. (The function-call operator is a unary operator.) If there are arguments, they should have spaces between them:

```
strong();
distance(rate, time);
weak(a, b, c);
funky(1, 5, c + d, e, f);
```

More will follow as we learn about more constructs.