C Coding Standards for EECS 381
Revised 2/24/09

Introduction

Each software organization will have its own coding standards or "style guide" for how code should be written for ease of reading and maintenance. You should expect to have to learn and follow the coding standards for whichever organization you find yourself in. For this course, you must follow this set of coding standards.

The Concept of Single Point of Maintenance

Programs get modified as they are developed, debugged, corrected, and revised to add new features. High-quality code makes modifications easier by having single points of maintenance instead of multiple places in the code that have to be changed.

Why program constants or parameters are named as symbols or const variables: Change the single definition, recompile, and the change takes place everywhere that name is used.

Why functions are used instead of duplicated code. Change the single function, rebuild, and that aspect of the program's behavior changes everywhere that function is used.

Many of these coding standards support single point of maintenance.

The Concept of Coding Reliability

Many bugs can be prevented by coding in a simple, clear, and consistent style that follows idioms and patterns that experienced programmers have developed.

Many of these coding standards improve coding reliability.

C compiler options for this course

Set compiler options to require function prototypes, the 1989 ANSI Standard, and pedantic warnings (for gcc).

Numeric types

Avoid declaring or using unsigned integers; they are seriously error prone and have no compensating advantage.

While they can never be negative, nothing prevents computing a value that is mathematically negative, but gets represented as a large positive value instead.

E.g `for(unsigned int i = 0; i < n; ++i)` is pointless and does nothing helpful and just makes a subtraction error possible.

If a number should never be negative, either test it explicitly, or document as an invariant with an assertion.

Major exception (not relevant to this course): if bitwise manipulations need to be done (e.g. for hardware control) using unsigned ints for the bit patterns may be more consistent across platforms than signed ints.

To interface with Standard Library functions, use size_t or casts; do not explicitly declare an unsigned integer type.

The only case for using unsigned integers is to interface with Standard Library functions that return size_t values that traditionally are defined as unsigned to allow a larger possible size. But never declare such variables as "unsigned int"; instead:

Declare and use size_t variables to hold the values, as in:

```
size_t len = strlen(s);
```

or (preferred) cast between size_t values and int values, as in:

```
int len = (int) strlen(s);
```

Prefer double to float.

Only use of float: if memory space needs to be saved, or required by an API.

Prefer double to float.

Note that in C89 (at least) float argument values are automatically converted to double in function calls.

Do not assume that two float or double values will compare equal even if mathematically they should.

Only case where they will: small to moderately large integer values have been assigned (as opposed to computed and then assigned).

Otherwise, code should test for a range of values rather than strict equality.

String literal constants

Prefer to declare and define as pointers to constants rather than initialized arrays of char:

Bad: `const char message[] = "Goodbye, cruel world!";`

Requires extra storage for the array, plus time to copy the literal into the array.

message is an array sized big enough to hold the string which is copied in at initialization, even though the string has already been stored in memory.

Good: `const char * const message = "Goodbye, cruel world!";`

Simply sets a pointer to the string literal already stored in memory.

message is a constant pointer to constant characters - neither the pointer nor the characters can be changed.

Enum types

Prefer to use an enumerated type instead of arbitrary numeric code values.

The names in the enumeration express the meaning directly and clearly.

Do not use an enumeration if the result is greater program complexity.

E.g. translating command strings into enums which are then used to select the relevant code for the command simply doubles the complexity of command-selection code.

Give the type a name starting with an upper-case letter terminated with "_e" or similar.

The names for the values should be all upper case.

Always store and pass enum values in the enum type, even though they convert freely to ints.

Keep it in the enum type to maintain the clear and direct meaning.

Prefer to use the default for how enum values are assigned by the compiler

Bad: `enum Fruit_e {APPLE = 0, ORANGE, PEAR, BANANA};// Why? This is the default!`

Not relying on the default when it is suitable indicates either ignorance or confusion.

Bad: `enum Fruit_e {APPLE = 3, ORANGE, PEAR, BANANA};// Potential fatal confusion!`

There needs to be a VERY GOOD reason to override the compiler-assigned values.

Good: `enum Fruit_e {APPLE, ORANGE, PEAR, BANANA};// Let the compiler keep track!`

Understand and use how I/O works with enum values

Enums are written as integer values

To read an enum value, read an int, check if for the maximum and minimum valid values, and then assign with a cast to the enumerated type.

Names

Take names seriously - they are a major way to communicate your design intent to the future human reader (either yourself or somebody else).

Poor names are a major obstacle to understanding code.

Do not use cute or humorous names, especially if they don't help communicate the purpose of the code.

Bad: `delete victim; // there is no "victim" here`

Do not use cute or humorous names, especially if they don't help communicate the purpose of the code.

> Better: `delete node_ptr;  // there is a node that we are deleting`
>
> Bad: `zap();    // sure, it's cute, but what does it do?`
>
> Better: `clear_pointers(); // ok - this function clears some pointers.`

Don't start variable or function names or #define symbols with underscores.

> Leading underscores are reserved for the implementation - break this rule, and you risk name collisions leading to confusing errors.

Use an initial upper-case name for your own types (enums, classes, structs, typedef names).

> e.g. `struct Thing`, not `struct thing`.
>
> Standard Library symbols are almost all initial lower-case, so this is an easy way to distinguish your types from Standard types.

Distinguish enum names with a final "_e", as in Fruits_e;

The values for an enum type must be all upper case.

Distinguish typedef names with a final "_t", as in Thing_list_t;

Preprocessor symbols defined with #define must be all upper case.

> Bad: `#define pi 3.14159265`
>
> Good: `#define PI 3.14159265`

Distinguish names for constants that are declared variables. Choose and maintain a style such as a final "_c" or a leading lower-case 'k' followed by an upper-case letter .

> Bad: `ymax = screen_h_size;    // no clue that right-hand-size is a constant`
>
> Good: `const int kScreen_h_size = 1024;`
>
> Good: `const int screen_h_size_c = 1024;`
>
> Good: `const char * const error_msg_c = "Error encountered!";`

Use typedef to provide a more meaningful, shorter, or detail-hiding name for a type.

> Little value if the typedef name is as verbose as the type.
>
> Bad: `typedef struct Thing * Thing_struct_ptr_t;`
>
> Good: `typedef struct Thing * Thing_ptr_t;`

Use variable names that do not have to be documented or explained - longer is usually better.

> Worst: `x;`
>
> Bad: `bsl;`
>
> Good: `binding_set_list;`

Single letter conventional variable names are OK for very local, temporary purposes.

> OK:
> ```
> for(int i = 0; i < n_elements; i++)
>     sum = x[i];
>
> y = m * x + b;
> ```

Don't ever use easily confused single-letter variable names - don't rely on the font!

> Lower-case L (l), upper-case i (I) are too easily confused with each other and the digit one.
>
> Similarly with upper-case O and digit zero.

Use upper/lower mixed case or underscores to improve readability of explanatory names.

> Bad: `void processallnonzerodata();`

Use upper/lower mixed case or underscores to improve readability of explanatory names.

Good: `void ProcessAllNonZeroData();`

Good: `void process_all_non_zero_data();`

Don't include implementation details such as variable type information in variable names - prefer to emphasize purpose instead.

Bad: `int count_int;`

Bad: `const char * ptr_to_const_chars;`

Better: `int count;`

Better: `const char * input_ptr;`

How to tell: What if I need to change the variable type to a similar but different type? E.g. long ints, wide characters. Would it be important to change the variable names to match? If so, implementation details are exposed in the variable names.

A name or symbol for a constant that is a simple synonym for the constant's value is stupid. The purpose of naming constants is to convey their role or meaning independently of their value; often, the same concept might have a different value at some point in the future.

See discussion of Macros and Single Point of Maintenance

Bad: `#define TWO 2  // what else would it be? 3? This is stupid!`

Bad: `#define X 4    // what is this? Can't tell from this name!`

Good: `#define MAX_INPUT_SIZE 255 // the maximum input size, currently this value`

Good: `const double aspect_ratio = 16./ 9.; // We can tell what this is!`

Global variables

In this course, global variables can be used only where specifically authorized.

Global variables should never be used simply to avoid defining function parameters.

Experience shows that passing information through parameters and returned values actually simplifies program design and debugging - global variables used for this purpose are a common source of difficult-to-find bugs.

Global variables are acceptable only when they substantially simplify the information handling in a program. Specifically, they are acceptable only when:

Conceptually, only one instance of the variable makes sense - it is holding information that is unique and applicable to the entire program.

E.g. the standard I/O streams are global variables.

They have distinctive and meaningful names.

They are modified only in one or two conceptually obvious places, and are read-only elsewhere.

They are used at widely different points in a function call hierarchy, making passing the values via arguments or returned values extremely cumbersome.

Their linkage is carefully handled to avoid ambiguity and restrict access if possible.

i.e. C++ methodology is followed.

Internal linkage if possible.

Global constants defined as `const` variables at file-scope or externally linked (program-scope) are not global variables - these restrictions do not apply.

Read-only, with a single definition, does not present any maintenance or debugging problem, and helps ensure consistency.

Be sure they are fully non-modifiable - e.g. `const char * const` for pointers to string literals.

If file-scope, best to give them internal linkage.

Declare `static` in C.

If file-scope, best to give them internal linkage.

File-scope const variables are automatically internally linked by default in C++.

If program-scope, follow global variable guidelines:

Put `extern` declarations only in a header file.

Put definition and initialization in a .c or .cpp file that `#includes` the header file.

Macros (Preprocessor)

Avoid defining anything except the simplest macros.

Constants and include guards is about all that you should be doing in this course.

All symbols defined with #define must be ALL_UPPER_CASE.

A critical reminder that a macro is involved.

Define symbols for constants that might need to be changed, especially if they appear in multiple places in the program.

Avoid hard-coded "magic" numbers or other constants - a serious maintenance problem - you want a single point of maintenance for the whole program.

E.g. array sizes are especially important - often need to be changed as a group.

Bad:
```
...
char buffer[64];
...
char name[64];
...
... = malloc(64);        /* allocate memory for an input string •/
...
```
Good:
```
#define INPUT_LENGTH 64 /* a single point of maintenance */
...
char buffer[INPUT_LENGTH];
...
char name[INPUT_LENGTH];
...
... = malloc(INPUT_LENGTH);      /* allocate memory for an input string •/
...
```

Don't define symbols if they are simply synonyms for constants fixed in the language.

Criterion for a useful symbol: Could you meaningfully change the value without changing the name?

Stupid: What if you change the value?
```
#define THREE 3
```
Bad:
```
#define TRUE 1
#define FALSE 0
```
Some conditions and Standard function return values that are true in the non-zero sense will not necessary be == 1. So comparing them to a TRUE value of 1 would fail.
The true/false meaning of non-zero/zero is part of the definition of the language and should be used by the programmer.
Good:
```
#define INPUT_LENGTH 64
```

Use the assert macro liberally to help document invariants and help catch programming errors.

ONLY for programming errors - not for run-time errors to be reported to the user.

NEVER to test for successful resource allocation (e.g. from malloc).

errors.

Idiomatic C

Use NULL to refer to a pointer value of zero (in C only)

Note: the Standard definition of NULL is simply zero:

```
#define NULL 0
```

Take advantage of the definition of non-zero as true, zero as false, when testing pointers or integer values.

Clumsy:

```
if(ptr != NULL) or if(ptr == NULL)

if(flag != 0) or if(flag == 0)
```

Better:

```
if(ptr) or if(!ptr)

if(flag) or if(!flag)
```

Take advantage of how C-strings are designed to work well with pointers with the terminating null byte providing a built-in sentinel that can be tested by a loop condition.

Bad:

```
void string_copy(char * dest, const char * src)
{
    size_t i, len = strlen(src);
    for(i = 0; i < len; i++)
        dest[i] = src[i];
    dest[i] = '\0';
}
```

Good:

```
void string_copy(char * dest, const char * src)
{
    while(*dest++ = *src++);
}
```

Best: Use the Standard Library strcpy function - which may be optimized for the architecture.

Write for *statements in their conventional form if possible.*

Good - the conventional, most common form:

```
for(i = 0; i < n; i++)  // correct for almost all cases
```

Bad:

```
for(i = 1; i <= n; i++) // confusing - what is this about?
for(i = n; i > 0; i--)  // better be a good reason for this!
for(i = -1; i <= n; i++)    // totally confusing!
```

Designing functions

Use functions freely to improve the clarity and organization of the code.

Modern machines are very efficient for function calls, so avoiding function calls is rarely required for performance.

If it is, prefer inline functions to get both performance and clarity.

Define functions that correspond to the conceptual pieces of work to be done, even if only called once or from one place.

Clarify the code structure, making coding, debugging, maintenance, easier.

E.g. in a spell-checking program, create a function that processes a document by calling a function that processes a line of the document that in turn calls a function that finds each word in the line.

Use functions to avoid duplicating code.

Copy-paste coding means copy-pasting bugs and multiplying debugging and modification effort.

Concept: *Single point of maintenance*. If you have to debug or modify, you want one place to do it.

How do you tell whether duplicated code should be turned into a function? Move duplicated code into a function if:

The code is non-trivial - getting a single point of maintenance is likely to be worthwhile.

What the code does can be separated from the context - you can write a function with simple parameters and return value that does the work for each place the duplicated code appears.

The result is less total code with the complexity appearing only in a single place - the function - giving a single point of maintenance.

If functions are used (or should be used) only within a module, give them internal linkage and keep them out of the header file.

Do not put the prototype in a header file; declare and define in the .c file as static.

To tell the compiler you aren't using a function parameter in a definition, leave out its name.

```
void foo(int i, double x) { code that doesn't use x} – gets a warning about unused x

void foo(int i, double) { code } – NO warning about unused second parameter
```

For functions that might encounter an error condition, return a error value from the function, and the calling code should always check the returned value for the error value before continuing.

If a pointer is returned, a NULL value is usually the best error code value.

Otherwise, return an int, where zero means "no error" and non-zero means an error - this allows the cleanest checking code.

Avoid Swiss-Army functions that do different things based on a switching parameter.

Bad:
```
void use_tool(/* parameters */, int operation)
{
     if(operation == 1)
          /* act like a corkscrew */
     else if(operation == 2)
          /* act like a screwdriver */
     else if(operation == 3)
           /* act like a big knife */
     else if(operation == 4)
           /* act like a small knife */
etc
}
```

The problem is that you can't tell by reading the call what is going on - you have to know how the switching parameter works, and what the other parameters mean depending on the switching parameter, etc. Separate functions for separate operations are usually better.

especially bad if the resulting code is almost as long or longer than separate functions with good names would be.

using an enum for the switching parameter helps only slightly because it clutters the rest of the program.

Could be justified if:

the switch parameter is very simple (like true/false only)

the behavior controlled by the switching parameter is conceptually very simple (like turning output on or off)

the switched-function is considerably smaller, simpler, and re-uses code much better than separate functions would.

the function call is always commented with an explanation of the switching parameter value

Code structure

Put function prototypes and struct declarations in the header file if they are part of the module interface, or at the beginning of the implementation file if not.

This ensures that the function definitions can appear in a meaningful and human-readable order - e.g. from top-level down to lowest-level.

See Layout discussion.

Declare variables in the narrowest possible scope.

Note that {} defines a block with a new nested scope in if, while, for, do-while statements, or even all by itself.

Declare variables within a if/for/while block where possible.

But beware possible inefficiency if a function has to be called to initialize the variable.

If a group of variables is only needed in a portion of the code in a function, consider creating a block for that portion and declare the variables at the start of the block.

Especially if the function is long or complex.

Note that if the portion of the code is controlled by a conditional or iteration, the appropriate block might already be present.

Bad:
```
void foo()
{
    int i = 0;
    int j = 1;
    int k = 2;
    int m = 3;
    int n = 4;
    /* lots of code using only i, j, k */
    /* some code using m and n */
}
```
Better:
```
void foo()
{
    int i = 0;
    int j = 1;
    int k = 2;
    /* lots of code using only i, j, k */
    {
        int m = 3;
        int n = 4;
        /* some code using m and n */
    }
}
```
Prefer "flat" code to deeply nested code.

Deeply nested code is hard to read and fragile to modify. Prefer a code organization of a "flat" series of condition-controlled short code segments. This will usually require re-thinking the logic, but the result is simpler and easier to work with.

Bad:
```
if(...) {
    ...
    if(...) {
        ...
        if(...) {
            ...
            if(...) {
                ...
            }
```

```
                    ...
                }
            ...
        }
        ...   // I'm lost - just when does this code execute?
    }
```
Better:
```
if(...) {
        ...
        }
else if(...) {
        ...
        }
else if (...) {
        ...
        }
etc
```
The "single point of return" guideline usually results in deeply nested conditional code. Such code can usually be rewritten as a simple series of conditionals each controlling a block of code that ends with a return. This works especially well if the conditions are checking for error situations.
Usually good:
```
if(...) {
        ...
        return;
        }
if(...) {
        ...
        return;
        }
if (...) {
        ...
        return;
        }
...
return;
```
Prefer using a switch statement to if-else if constructions to select actions depending on the value of a single variable.

Generally results in simpler, clearer, and faster code than the equivalent series of if-else if statements.

Exceptions: switch statement cannot be used if strings or floating point values are being tested.

Not a good choice if ranges of integer values are being tested.

Always include a default case with an appropriate action (e.g. an error message or assertion).

Terminate each case with a break statement unless you deliberately want to arrange "drop through" to the next case; if so, you must comment on it.

Arrange iterative or performance-critical code to minimize function calls that might not be optimized away.

Be aware of what iterative code implies ... what has to be done each time around a loop?

Often, the compiler cannot tell whether the code will result in a function computing a different value during execution, and so will not attempt to replace multiple calls with a single call.
Bad: strlen gets called every time around the loop - and what does it do?
```
void make_upper(char * s)
{
    size_t i;
    for(i = 0; i < strlen(s); i++)
            s[i] = toupper(s[i]);
}
```

Better: compute the length of the string only once before starting the loop.
```
void make_upper(char * s)
{
     size_t i, n = strlen(s);
     for(i = 0; i < n; i++)
          s[i] = toupper(s[i]);
}
```
Best - take advantage of how C-strings work - no need to compute length of string;
we have to access each character anyway, so just stop at the end.
```
void make_upper(char * s)
{
     for(; *s; s++)
          *s = toupper(*s);
}
```
or
```
void make_upper(char * s)
{
     while(*s = toupper(*s))
          s++;
}
```

Organize input loops so that there is a only a single input statement, and its success/fail status controls the loop, and the results of the input are used only if the input operation was successful.

In C/C++ the idiom is to place the input operation in the condition of a while loop.

Do not control an input loop based only on detecting EOF.

Input might have failed for some other reason, but bogus results will still be used, and EOF may never happen in the situation.

Bad -
```
while (fscanf(infile, "%d", &x) != EOF) {
     /* use x */
     }
/* loop might never terminate! */
```
Good - use data only if read was successful; diagnose situation if not:
```
while (fscanf(infile, "%d", &x) == 1) {
     /* use x */
     }
if(!feof(infile))
     /* not end of file – something else was wrong */
```
If only character or string data is being read, normally only EOF will cause the input to fail, so separate check to diagnose EOF is optional.

Ensure that input or data brought into the program cannot overflow the array or memory block in which it is to be stored.

A basic and essential security and reliability precaution.

Assume user or file input can contain arbitrarily long random strings, and write code that can handle it safely and reliably, even if it simply ignores over-length input.
Use functions and facilities that specify explicit maximum lengths for input.

Using the Standard Library

Don't recode the wheel - know and use the Standard Library functions.

You can assume that the Standard Library is well-debugged and optimized for the platform.

If it seems at all likely that another programmer has needed what you need, look it up and see if it is in the Standard Library.

Unnecessary DIY coding wastes both coding time and debugging time.

E.g. why write, test, and debug code that reads characters until the first non-whitespace character and then reads and stores it, when scanf(" %c", &char_var); will do it for you?

If there is a reason why the obvious Standard Library function can not be used, comment your own function with an explanation.

Understand what Standard Library functions do, and trust them to do it correctly.

Don't waste time writing code that only makes sense if the Standard Library is defective.

Bad:
```c
char str[51];/* actually should be using #define symbols here to avoid magic numbers and strings */
int result;
for(i = 0; i < 51; i++)
     str[i] = '\0';   /* lets make sure str is completely empty, just in case ... */
result = scanf("%50s", str);
if(result != 1)  /* check and return an error code just in case this failed somehow, */
     return 1;
str[50] = '\0';  /* lets make sure str is properly terminated, just in case ... */
assert(strlen(str) <= 50);  // check that we didn't overflow the array, just in case ... */
```
Good:
```c
char str[51];/* actually should be using #define symbols here to avoid magic numbers and strings */
scanf("%50s", str); /* will always succeed in this course unless something is grossly wrong */
/* str is good to go */
```

Do not write functions that simply wrap a Standard Library function.

Assume that your reader is (or should be) familiar with the Standard Library; this means that calling the Standard Library function directly will be more comprehensible than figuring out your particular function that does little or nothing more than call the Standard function.

Bad: /* with reader's comments shown */
```c
...
     int i;
     if(read_int(&i)) { /* uh ... exactly what does that function do? */
...
/* let's find the function definition and check it out */

int read_int(int * iptr) {
     int result;
     result = scanf("%d", iptr);
     return result == 1;
}
/* gee - doesn't really do anything! */
/* why did the programmer bother with this function? */
```
Good:
```c
...
     int i;
     if(scanf("%d", &i) == 1) {  /* no problem understanding this */
...
```

Do not use the memmove/memcpy/memset family of functions in this course.

Unless the rest of the code is completely free of inefficiency - "lipstick on a pig" otherwise.

Using dynamically allocated memory (malloc)

Where possible, use "automatic" function-local variables. Do not allocate memory with `malloc` *if a local variable or array will work just as well.*

Do not apply a cast to the `malloc` *call.*

Modern `malloc` returns a `void*` which can be assigned to any pointer type, by definition.

The traditional cast is an anchronism left from pre-`void*` days, when `malloc` returned a `char*`. The cast suppresses an important warning if you forget to `#include <stdlib.h>`.
Bad:
```
int * p = (int *) malloc(n * sizeof(int));
```
Good:
```
int * p = malloc(n * sizeof(int));
```

Always check the returned value of `malloc` *family functions to verify that memory allocation was successful before using the returned value.*

Take some positive action such as printing a message and calling `exit`.

Design your code with a clear and explicit policy that states where and when the call to `free` *will be for every call to* `malloc`.

Attempt to write the deallocation code immediately after the allocation code to avoid forgetting it.

In this course, all allocated memory must be deallocated by the program before terminating.

Represents the "clean up before quitting" philosophy - a good practice even if often not strictly necessary in modern OS environments.
Program must terminate with a return from `main` after deallocating all memory.

Only permitted exception: when the `exit` function is called because of memory allocation failure.

In this course, use only `malloc` *for allocating memory; do not use the other functions.*

Other Library functions such as `calloc` or `realloc` may not be used. Also, you may not use non-Standard library functions such as `strdup` that allocate memory.
Functions such as `calloc` and `realloc` are occasionally useful, but have more complex interfaces, hide some important details, and do very little more than you can do with your own code and `malloc`. `malloc` is the simplest and most basic memory allocation function, and so you will understand the concepts better by using only it to allocate memory.

Header files should be a minimal declaration of the module interface.

See the header file guidelines document for more discussion and detail.

Program modules or re-usable components consist of a header (.h) file and an implementation (.c) file.

The header file should contain exactly the interface declarations required for another module (the client) to use the module or component, and no more.

Any declarations or definitions not strictly required as part of the interface should be in the implementation file, not the header file.

Arrange to have the minimum number of #includes in a header file.

Use forward/incomplete declarations of pointer types instead of #includes if possible.

The header file should be complete; it should compile correctly by itself.

> Create a .c file that contains nothing but an #include of the header file. This file should compile without errors.

Guidelines for #including header files in an implementation file.

The first #include in a .c file for a module should be the corresponding header file.

Project-specific includes (using double quotes) should appear before Standard Library or system includes (with angle brackets).

Always ensure that the relevant Standard Library header gets included even if the code happens to compile without it.

> This prevents platform-specific compile failures due to how the Standard doesn't say which Library headers have to include which other Library headers.

Do not #include unnecessary header files.

> Causes serious problems with spurious coupling and slower compile times.

Using a project Utilities module

Place in the Utilities module only functions or declarations that are used by more than one module.

> Examples: A typedef used throughout a project; a function that compares two struct type variables that is needed in two modules.

Secondarily, place in the Utilities module functions that are generic and would be generally useful in related projects.

> Examples: a function to convert 12-hour time to 24-hour time; a function to produce a lower-cased copy of a string.
>
> Negative example: a function that isolates words in a string following the rules for a particular spell-checking implementation.

Do NOT use the Utilities module as a dumping ground for miscellaneous scraps of code - it is reserved for the above two uses.

> Project-specific code used by only one module should never be placed in the Utilities module.

Layout

Arrange function definitions in a .c file in a human-readable order corresponding to the top-down functional decomposition or usage order of the module.

The reader should be able to read the code in increasing order of detail to take advantage of the information-hiding value of functions. So the root(s) for the function call tree should be the first functions listed; leaf functions called only from one branch should appear before the start of the next branch; leaf functions called from all branches should appear last.

Don't make the reader rummage through the file trying to find functions listed in a haphazard order.

Use a consistent indenting scheme and curly brace scheme.

Imitating Kernigan & Ritchie or Stroustrup is certainly one good approach.

Avoid excessively long lines - 80 characters is a traditional value.

If lines won't fit on standard paper when printed in 10 pt font, they are too long.

Especially bad: long lines due to excessively nested code, which has other serious problems.

Be careful with leaving out optional curly braces, especially with if.

Clear: a simple thing that also looks simple:
```
if(x == 3)
      foo(x);
```
But if we later add some more code to the if, it is just too easy to write:
```
if(x == 3)
      foo(x);
      zap();      /* uh ... why doesn't it work right? */
```
Uglier but more reliable when coding late at night:
```
if(x == 3) {
      foo(x);
      }
```

Comments

See the posted article on comments for more discussion and examples.

Keep comments up-to-date; at least annotate or delete them if they are no longer valid.

Obsolete comments suggest sloppy coding at best, and are often worse than none at all because they confuse and mislead, and cast doubt on all of the other comments. Out-of-date comments will be considered a major failure of code quality.

Comments should never simply paraphrase the code.

You should assume that the reader knows the language at least as well as you do. The purpose of comments is to explain aspects of the code that will not be obvious to an experienced programmer just by looking at it.

Each function prototype in a header file, and function definition in a .c file, should be preceded by a comment that states the purpose of the function and explains what it does.

The function name and parameter names should well chosen, which will help explain how the function is used. If a value is returned, it is important to explain how it is determined - this will usually be less obvious than the role of well-named parameters.

In a .c file that has a block of function prototypes at the beginning, comments are not required for the function prototypes, but are required on the function definitions.

The initial prototypes are declarations for the compiler, and enable the functions to be defined in a readable order, but the prototypes are inconveniently located for the human reader - comments there are wasted.

The purpose of constants should be commented, especially if they may need to be changed.

E.g. a constant for the maximum length of an input line.

Comments should appear within a function to explain code whose purpose or operation is obscure or just not obvious.

Comments should explain what is being done where the code will be less than completely obvious to the reader. A common student error is comment simple code, but then make no comment at all to explain a chunk of difficult and complicated code that obviously took a lot of work to get right. If it was hard for you to write, it will be hard for a reader to understand!