A note on editors: No one cares what editor you use. Arguments over the best or most superior editor are a waste of time and energy. Use the editor that you like and is available on the system that you are using. Any mention of an editor in this document is illustrative only and not intended to promote one editor over another. The same holds for IDEs and the IDE v. command line debates. If you simply have to win some editor war, claim to do all your programming with one command: cat > /dev/mem.

Indentation

A consistent approach to indentation is essential. Consistent indentation makes the code easier to read for the human reader; the compiler doesn’t care. There are many approaches to indentation, each of which has its own strengths and weaknesses.

1. When modifying existing code, use the existing indentation style. Mixing styles is a sure way to confuse readers, waste time, and increase the chance of bugs being introduced to the code.

2. All lines should be shorter than 80 characters, preferably 72 characters or less. This helps with both printing and viewing code – don’t assume that others use the same font or window size as you!

3. Using the tab character is a common way to indent code. There are caveats:
   
   (a) Not all programmers use the same tab length.
      i. This can be set in most editors.
      ii. 8 spaces is pretty standard.
      iii. The TAB character will be inserted, not spaces!
   
   (b) It is common for programmers to change the tab length inside their editor. This doesn't change it for any other purpose and programmers often get surprised when their printouts are unreadable.

   (c) The original use of the TAB character was due to much smaller storage in the past.

4. 4 spaces is also common.

5. 2 spaces follows the K&R style but can be harder to read.

   examples:
   
   // 2 spaces
   {
   while ( i < max_line_length ) {
       if ( foo == bar ) {
           printf("This is a message to be printed. Not overly long, but ...");
           // some code
       }
   }
   // some code
6. Modern editors will have some form of “smart” indentation based on the programming language.

7. Modern editors will allow the programmer to set the number of spaces for indentation.

An example entry for the .vimrc file under Linux:

```
:filetype plugin indent on
:set smartindent
:set cino=>2   " 2 spaces for indentation
```

## Names

1. Take names seriously – they are a major way to communicate your design intent to the reader.

2. Poor names are a major obstacle to understanding code and can hide intent.

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

   (a) Bad: delete victim; // there is no "victim" here
(b) Better: `delete node_ptr; // there is a node that we are deleting`
(c) Bad: `zap(); // sure, it’s cute, but what does it do?`
(d) Better: `clear_pointers(); // ok - this function clears some pointers.`

4. Use upper/lower mixed case or underscores to improve readability of explanatory names.
   (a) Bad: `void processallnonzerodata();`
   (b) Good: `void ProcessAllNonZeroData();`
   (c) Good: `void process_all_non_zero_data();`

5. 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.

6. Use an initial upper-case name for your own types (enums, classes, structs, typedef names).
   `struct Thing, not struct thing.`

7. Standard library symbols are almost all initial lower-case, so this is an easy way to distinguish your types from standard types.
   (a) The initial upper-case makes the distinction between type names and variable names obvious.
   (b) Distinguish enum names with a final "e", as in `Fruits_e;`
   (c) The values for an enum type must be all upper case. Treat as constants.
   (d) Distinguish typedef names with a final "t", as in `Thing_list_t;`
   (e) Preprocessor symbols defined with #define must be all upper case.
      i. Bad: `#define collection_h`
      ii. Good: `#define COLLECTION_H`

8. Use variable names that do not have to be documented or explained – longer is usually better.
   (a) Worst: `x;`
   (b) Bad: `bsl;`
   (c) Good: `binding_set_list;`
   (d) Single letter conventional variable names are OK for temporary purposes.
      ```
      for(int i = 0; i < n_elements; i++) // OK use
          sum = x[i];
      y = m * x + b;
      ```

9. Don’t use easily confused single-letter variable names – don’t rely on the font!
   (a) Lower-case L (l), upper-case i (I), and the digit one (1) are too easily confused.
   (b) Similarly with upper-case O and digit zero.

10. Don’t include implementation details such as variable type information in variable names – prefer to emphasize purpose instead.
(a) Bad: `int count_int;`
(b) Bad: `const char * ptr_to_const_chars;`
(c) Better: `int count;`
(d) 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.

**Numeric Types**

1. Use the correct type.
   ```
   size_t len = strlen(s);
   ```
   Be sure you understand the limitations on the type. `size_t` is unsigned, but do you know why? Only cast between types if you have a need.
   ```
   int len = (int) strlen(s); // only if len could be negative
   ```

2. Unsigned integers can never be negative, nothing prevents computing a value that is mathematically negative but gets represented as a large positive value instead. Overflow becomes harder to detect.
   ```
   for (unsigned int i = 0; i < n; ++i) // overflow?
   for (unsigned int i = len; i >= 0; --i) // infinite loop
   ```

3. To interface with standard library functions, use the types of the function prototype as defined.

**Enumerated types**

1. Do not covert to integers, etc. Keep in the enum type to maintain the clear and direct meaning.

2. Let the compiler assign enum values
   ```
   Bad: enum Fruit_e APPLE = 0, ORANGE, PEAR, BANANA; // Why? This is the default!
   Rely on the name, not the value.
   ```
   ```
   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!
   ```

3. Don’t do arithmetic to determine an enum value – this undermines the purpose of the enum type and results in obscure and error-prone code. Often a switch statement is a better choice, especially if the enum value represents a state.

   ```
   Bad:
   ```
State_e new_state = old_state + 1;  // what does this mean? Can it go out of range?

Good:
State_e new_state;
switch(old_state) {  // obvious what is happening here
    case START:
        new_state = INITIALIZE;
        break;
    case INITIALIZE:
        new_state = OPEN_CONNECTION;
        break;
    case OPEN_CONNECTION:
        new_state = START_TRANSMISSION;
        break;
    // etc
}

Numerical Constants and “Magic Numbers”

1. Numerical or string constants that are "hard coded" or "magic numbers" that are written directly in the code are almost always a bad idea, especially if they appear more than once.

2. No single point of maintenance: If the value needs to be changed, you have to try to find every appearance in the code and fix it. If the value isn’t unique or distinctive, and appears many times, you are almost guaranteed to mess up a change!

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 — off by 1 error */
...

Good:
#define INPUT_SIZE 64 /* a single point of maintenance */
...
char buffer[INPUT_SIZE+1];
...
char name[INPUT_SIZE+1];
...
... = malloc(INPUT_SIZE+1); /* allocate memory for an input string */
3. Lack of clarity: Why does this naked number mean? What is its role? Why does it have the value it does? Could it change?
   
   Bad: horizontal = 1.33 * vertical;
   
   Good: horizontal = ASPECT_RATIO * vertical;

4. Exceptions: When a hard-coded value is acceptable or even preferable to a named constant:
   
   If the value is set "by definition" and it can’t possibly be anything else, and it has no meaning or conventional name apart from the expression in which it appears, then giving it a name as a constant can be unnecessary or confusing. Constants that appear in mathematical expressions are an example, but notice that some such constants have conventional names in mathematics and those names should be used.
   
   Example - notice that there is no conventional name for the value of 2 in this formula, but it can’t have any other value - giving it a name is pointless. In contrast, the value of pi is fixed but has a conventional name:
   
   \[
   \text{circumference} = 2.0 \times \text{PI} \times \text{radius}; /* 2 can only be 2 */.
   \]

   double degrees_to_radians(double degrees)
   {
      /* 180 degrees in a half-circle by definition */
      return degrees * PI / 180.;
   }

5. In some situations, a string literal is a “magic string” in that it might have to be changed for correct program functioning (e.g. a string containing a scanf format). However, output text strings can help make code understandable if it appears as literal strings in place rather than in a block of named constants.

6. A name or symbol for a constant that is a simple synonym for the value of the constant is pointless.

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

8. Bad: #define TWO 2 // what else could it be? 3? Don’t do this.
   
   (a) Bad: #define X 4 // what is this? Can’t tell from this name!
   (b) Good: #define MAX_INPUT_SIZE 255 // the maximum input size, currently this value
   (c) Good: #define ASPECT_RATIO = 16./9.; // We can tell what this is!

9. Preprocessor symbols defined with #define must be all upper case.
   
   (a) Bad: #define collection_h
   (b) Good: #define COLLECTION_H
   (c) Bad: #define pi 3.14159265
   (d) Good: #define PI 3.14159265

Global Variables

Definition: A global variable is any variable that is declared outside of a function - its scope is either the remainder of the file in which the declaration appears (internal linkage, e.g. “static”) or program-wide (external linkage). In either case, it is a global variable.
1. Global variables should never be used simply to avoid defining function parameters.

2. 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.

3. Global variables are acceptable only when they substantially simplify the information handling in a program. Specifically, they are acceptable when:
   (a) Conceptually, only one instance of the variable makes sense – it is holding information that is unique and applicable to the entire program.
   (b) They have distinctive and meaningful names.
   (c) They are modified only in one or two conceptually obvious places and are read-only elsewhere.
   (d) They are used at widely different points in a function call hierarchy; making passing the values via arguments or returned values extremely cumbersome.
   (e) Their linkage is carefully handled to avoid ambiguity and restrict access if possible.

4. Shadowing, when a variable declared within a certain scope has the same name as a variable declared in an outer scope, should never be used. It causes confusion and can result in serious bugs that can be very difficult to track down.

String Literals

1. String literals can be given names either as a preprocessor symbol with #define or as a const char * variable declared and initialized at file scope.

   ```
   #define PROMPT "Enter command"
   const char *prompt_msg = "Enter command";
   ```

2. 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 *message = "Goodbye, cruel world!";
   ```
   Simply sets a pointer to the string literal already stored in memory.

Idiomatic C

1. Use the standard macro NULL to refer to a pointer value of zero. In early C, there was no such definition; it has since been added to the language.

2. Take advantage of how C-strings are designed to work with pointers where the terminating null byte provides 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++);
}

3. Best: Use the Standard library strncpy function – which may be optimized for the architecture. Never use strcpy.

4. Write for statements in their conventional form if possible.
   Good – the conventional, most common form:
   
   \texttt{for(i = 0; i < n; i++) // correct for almost all cases}

   Bad:
   \texttt{for(i = 1; i <= n; i++) // confusing – what is this about?}
   \texttt{for(i = n; i > 0; i--) // better be a good reason for this!}
   \texttt{for(i = -1; i <= n; i++) // totally confusing!}

5. NEVER test for successful resource allocation (e.g. from malloc); check for FAILURE instead.

Designing Functions

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

2. Modern machines are very efficient for function calls, so avoiding function calls is rarely required for performance.
   
   If necessary, use inline functions to get both performance and clarity.

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

   This clarifies 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.

4. Use functions to avoid duplicating code.
(a) Copy-paste coding means copy-pasting bugs and multiplying debugging and modification effort.

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

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

   i. The code is non-trivial – getting a single point of maintenance is likely to be worthwhile.
   ii. 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.
   iii. The result is less total code with the complexity appearing only in a single location, giving a single point of maintenance.

5. Putting code in a function should add value to the code in some way, either by avoiding code duplication, hiding implementation details, or expressing a concept about the program behavior. If there are no details to put in one place or to hide, or concept being expressed, the function is distracting and pointless. Why make the reader find the definition just to discover the same code could have been written in place with no problem?

   Bad – no value added by the presence of a function compared to writing the same code in place:

   ```c
   void print_not_found_error_message ( void )
   {
     printf( "Specified data item was not found\n" );
   }
   ```

   Not quite as bad – a little bit of value added, by calling a cleanup function, but still very little value added:

   ```c
   void print_not_found_error_message ( void )
   {
     printf( "Specified data item was not found\n" );
     clear_rest_of_input_line ( );
   }
   ```

   Better – a general function that expresses a concept for how all error messages will be handled:

   ```c
   void print_error_message ( const char * msg_ptr )
   {
     printf( "%s\n", msg_ptr );
     clear_rest_of_input_line ( );
   }
   ```

   6. If functions are used (or should be used) only within a module, declare them **static** and keep their declarations out of the header file.

   7. 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 simplest and cleanest checking code.

```c
if (do_something()) {
    /* handle the error here */
}
/* OK if here */
```

**Code Structure**

1. Declare variables in the narrowest possible scope.
   
   Note that `{}` defines a block with a new nested scope.

2. Prefer "flat" code to nested code.

3. The misguided "single point of return" guideline usually results in deeply nested conditional code.

4. 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:

   ```c
   if (...)
   {
     ...
     return;
   }
   if (...)
   {
     ...
     return;
   }
   if (...)
   {
     ...
     return;
   }
   ...
   return;
   ```

5. 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?

6. 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]);
}

7. 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++;
}

8. Never modify the looping variable in the body of the loop.

9. 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.

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

11. A "priming read" - where the input operation appears twice, once before the loop, and then again in the
    loop – is a non-idiomatic anachronism from other languages - do not use in C or C++ code. Instead, use
    a do-while loop.

12. Validate input before using the data.

13. Note that testing the return value of fscanf for non-zero will not distinguish EOF or a partial success from
    a fully successful read.
14. If using scanf or fscanf, simply compare the returned value to the number of format items; if equal, read succeeded; if unequal, read failed.

15. 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.

16. Use functions and facilities that specify explicit maximum lengths for input.

Using the Standard Library

Deprecated Calls Many library functions exist for backwards compatibility. Read the manual page and if a function is deprecated do not use it. It is deprecated for a reason.

1. Don’t recode the wheel – know and use the standard library functions.

2. You can assume that the standard library is well-debugged and optimized for the platform.

3. 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. Learns to use man -k.

4. Unnecessary DIY coding wastes both coding time and debugging time.

5. If there is a reason why the obvious standard library function cannot be used, comment your own function with an explanation.

6. 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: (hard-coded constants used for brevity)

```c
char str[51];
int result;
    for (i = 0; i < 51; i++)
        str[i] = '\0';
    result = scanf("%50s", str);
    if (result != 1)
        return 1;
    str[50] = '\0';
    assert(strlen(str) <= 50);
```

Good: (hard-coded constants used for brevity)

```c
char str[51];
    scanf("%50s", str);
    /* str is good to go */
```

7. Assume that your reader is (or should be) familiar with the standard library.
8. Avoid functions that add no value.

Do not write functions that do nothing more than simply wrap a standard library function.

Bad: /* with reader's reactions shown */

... 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:

... int i;
   if(scanf("%d", &i) == 1) { /* no problem understanding this */
      ...

Using Dynamically Allocated Memory (malloc)

1. Do not apply a cast to the malloc call.
   C89 malloc returns a void* which in C can be assigned to any pointer type, by definition.
   The traditional cast is an anachronism left from pre-void* days, when malloc returned a char*. In C89, 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));

2. Always check the returned value of malloc for success before using the returned value.
   Take some positive action such as printing a message and calling exit.

3. Free dynamically allocated memory when it is no longer needed. Do not wait until the end of the program.

Layout

1. 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.
2. Use function prototypes to avoid awkward function order dependencies.

3. Use a consistent indenting scheme and curly brace scheme.
   
   Imitating Kernigan & Ritchie is one good approach.

4. Avoid excessively long lines – 80 characters is a traditional maximum value.
   
   If lines won’t fit on standard paper when printed in 10 pt font, they are too long.
   
   Long lines due to excessively nested code likely has other serious problems.

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

   Clear: a simple thing that also looks simple:
   
   ```c
   if (x == 3)
   foo(x);
   ```

   But if we later add some more code to the if, it is just too easy to write:
   
   ```c
   if (x == 3)
   foo(x);
   zap(); /* uh ... why doesn't it work right? */
   ```

   Uglier but more reliable when coding late at night:
   
   ```c
   if (x == 3) {
   foo(x);
   }
   ```

6. In C, indentation does not equal intent.

**Comments**

1. Do not comment what should be obvious from the (hopefully well written) code. See “Names”.

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

3. Obsolete comments suggest sloppy coding at best and are often worse than none at all because they confuse, mislead, and cast doubt on all of the other comments.

4. Comments are for human readers, not the compiler, so place comments where they are most convenient and useful for the human who is looking at the code.

   This is almost always at the function definition, rather than the function declarations.

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

5. 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.
6. Comments should appear within a function to explain code whose purpose or operation is obscure or just not obvious.

7. If you have more comments than code, rethink both.