**A MEMORY DUMP OF C PROGRAMMING,** *by Sean Eyre (@oni49) and Stephen Semmelroth (@diodepack) of Rainier Cyber*. This document is intended to provide you a quick desk reference for C's syntax and semantics. We included several example programs that will elaborate on each section in greater detail.
For the example files, see: https://github.com/Rainier-Cyber/C-cheatsheet

==================================================================

## File types

Write your program goes inside a file ending in ".c". For example, helloworld.c
"Headers" are files that your program imports and must end in ".h". For example, string.h

==================================================================

## Program structure (see program_structure.c)

The following block demonstrates the overall structure of the program, for more details on individual elements, see the rest of this document.

```
// Include external libraries. The // indicates a comment
#include <stdio.h>
#include <stdlib.h>

// Declare functions here for use later
int fxn(int a){ multiple statements; }  // See function section

// Program execution starts here.
int main(int argc, char **argv){
   fxn(13); // use declared functions
   printf("Example program structure.\n");  // See Input/Output
   return 0; // return 0 to indicate success
}
```

`main()` is the function that will be called by the system when the program executes. It must always return an int representing the success or state of failure the program is in upon exiting. If the program executes completely and successfully, it will return 0, which is an integer.

==================================================================

## Statements (Sequencing)

Statements end with a ";" (semicolon) and perform a basic action. They are performed in the order the appear. Usually, only one statement appears on one line. However, when similar functions are grouped together, your organization's style guide may require you to put them on the same line.

```
a = 3 + 10;  // A single line statement

// Two statements on one line
char *y; y = malloc( sizeof( char ) * length );
int x, y, z; // Equivalent to int x; int y; int z;

// Split line statement
printf("I'm a really long statement printing %d, %d, %d, and %d.\n",
       5, 4, 3, 1);  // Whitespace ignored because partition by ( and )
```

Throughout most of this document's examples, `{ multiple statements; }` can also be substituted for `stmt;`

==================================================================

**Operators** The chart below is presented in order of operations. Which means, the operators at the top of this chart are performed before the operators at the bottom. For example, in `a == !b`, the NOT operator is applied first, followed by the equality operator. Or for `a / (int) b`, b is first cast to an integer before `a / b` is evaluated.

| Operator | Description |
|---|---|
| *struct.member* | **Structure member reference** |
| *pointer->member* | **Structure pointer** |
| `!; ~` | **Logical NOT, Bitwise NOT** |
| `++; --` | **Inc/decrement** |
| **pointer; &name* | **Indirection via pointer, address an object** |
| *(type) expr* | **Cast *expr* to *type*** |
| `sizeof` | **Size of a double in addressable bytes** |
| `*; /; %` | **Multiplication/ Division/ Modulo (checks for remainder after division)** |
| `+; -` | **Addition/ Subtraction** |
| `<<; >>` | **Bit shift left/right; each shift is x2 or /2** |
| `<; >` | **Less/greater than** |
| `<=; >=;` | **Less/greater than or equal** |
| `==; !=` | **Equal/ not equal to** |
| `&` | **Bitwise AND; 1 & 1 = 1, 1 & 0 = 0, 0 & 0 = 0** |
| `^` | **Bitwise XOR; 1 ^ 1 = 0, 1 ^ 0 = 1, 0 ^ 0 = 0** |
| `\|` | **Bitwise OR; 1 \| 1 = 1, 1 \| 0 = 1, 0 \| 0 = 0** |
| `&&` | **Logical AND** |
| `\|\|` | **Logical OR** |
| `?:` | **If-like expression** |
| `=; +=; -=; *=` | **Assignment; modify and assign** |
| `,` | **Evaluations separator** |

Operator Examples (see operators.c)

```
4 << 2;  // Shift decimal 4 ("100" in binary) two bits left;
         // Returns "10000" or 16 in decimal; Single shift left is a x2
         //   Single shift right is a /2
( x % 2 == 0 ) ? "Even" : "Odd";
int a = 1, b = 2, c = 3;  // Link together like statements with ","
```

==================================================================

## Conditional Statements (Selection, see example conditionals.c)

```
if ( cond ) stmt;  // Execute stmt if cond is true, single-line if

// Exec stmt if true, stmt_2 if false, single-line if-else
if ( cond ) stmt; else stmt_2;  // Pay attention to semicolons

if ( cond ) {  // Execute code block if true
   stmt;
   stmt;
}  // No semicolon here!

// If-else block
if ( cond ) stmt;
else if ( cond_2 ) stmt_2;
else if ( cond_3 ) { multiple statements; }
else stmt_4;
```

Switches can be used in place of if-else when selecting based on a discrete constant (ex, an unchanged integer) (example switches.c)

```
switch ( cond ) {
   case const_1: stmt; break;  // Breaks prevent fall..
   case const_2: stmt; break;  // ..through cases/errors
   default: stmt;  // Default case is the final "else"
}
```

=================================================================

**Loops (Iteration,** see example loops.c**)**

`{ multiple statements; }` can also be substituted for `stmt;` in all the below.

```
while ( cond ) stmt;  // Execute stmt only if cond true

// Execute stmt then do again if cond true
do { stmt; } while ( cond )  // Always use {} even if on same line
```

For-loops start at *init* and continue until *cond* becomes true. After each execution, *init* is changed by the process indicated in *step*

```
for ( init; cond; step) stmt;
for (i = 0; i < 10; i++) printf("%d", i);
```

=================================================================

**Key words**. Compiler protected words. They cannot be used as variable or function names.

```
auto            double          int             struct
break           else            long            switch
case            enum            register        typedef
char            extern          return          union
const           float           short           unsigned
continue        for             signed          void
default         goto            sizeof          volatile
do              if              static          while
```

=================================================================

**Data types and Declaring variable.** (see example variables.c)

| Name | Size | Desc |
|---|---|---|
| `char` | 1 byte | A single ASCII value like 'a', 'b', '1', '2', '&' |
| `int/long` | 4 bytes | A signed integer; positive, zero, or negative whole number |
| `long long` | 8 bytes | A longer, multi-byte signed integer |
| `float` | 4 bytes | A floating-point value; a real number/decimal value |
| `double` | 8 bytes | A double length float |

Notice that "strings" are not in the above list. That's because strings in C are static **ARRAYS OF CHARACTERS**—they do not change length unless memory is reallocated.

Zero values are logical false, nonzero are logical true. `0 == False, -42 == True, 1 == True` . This allows you to perform selection directly on integers instead of creating and using Booleans. Since processes return integers when they exit, it also allows you to perform selection based on the process exit state.

Variable names must begin with a lowercase letter, be alphanumeric, and not be a key word.

```
type name = value;  // Declare a variable of type
type name[n] = {value1, ...};  // type array of size n
char name[7] = "string";  // Declare a string of size 6 + null term

int a[10];  // An array of 10 ints (int is the data type stored)
int i = a[6];  // This sets i to the 6th element of a
// The address of array a is the same as the address of a[0]
// Array indexes begin at zero
int *p = a;  // Pointer to address of a, same as &a[0] (see & operator)
```

Strings are *arrays* of characters that terminate with `\0` (`null terminator`, which uses an index), they are not their own true data type. Make sure to include string.h—common string utilities are found in string.h, notably the following (see example strings.c)

```
char s[10] = "my string"; // String declaration includes room for \0
strcpy(d, s)  // Copy s to d, does not check memory boundaries, *avoid*
              // because it is unsafe
strncpy(d, s, n)  // Copy at most n char s to d, stops at boundaries
memcpy(d, s, n)  // Copy n BYTES s to d, no boundaries check
memmove(d, s, n)  // Copy n BYTES, won't fail if exceeds boundaries
```

You <u>don't have to</u> include string.h in order to use strings, however it provides utilities that will substantially ease your pain. Whenever a function does not check boundaries, using a source (s) with more bytes than the destination (d) will cause an overflow error. Your program will still compile and your compiler <u>might</u> warn you about the error. The program might still execute properly, but most of the time you will have introduced undefined behavior and an opportunity for exploitation—this is often the case when people use strcpy().

Qualifiers and Storage Classes:

```
const type var_name;  // Flag variable as read-only
volatile type var_name;  // Flag variable as unpredictable
register type var_name;  // Quick-access via RAM or register,
                         //   cannot exceed register size
static type var_name;  // Variable value preserved btwn fxn calls
extern type var_name;  // The variable is declared by another file
```

=================================================================

**Functions** (example functions.c)

Functions point to a block of code in memory and are called with the function's name and parameters. Functions have a return a data type or are void and return nothing.

```
ret_type name(type param_1, type param_2, ...){  // Declare function
        Multiple statements;
        return x;  // Same type ret_type above, often int/ bool/ char
}
```

For example, this function is named "add_two" and expects the user to pass it an integer. It will return another integer.

```
int add_two(int n) {
   int x = n + 2;
   return x;
}
```

Parameters all have types. When the function is called, the computer instantiates a variable (allocate memory) within the function’s scope (memory space). That variable is initialized by the value passed to the corresponding parameter when the function is called. After returning, the program will dereference the function’s variables, unless they are static.

Function names without parameters return a pointer to the function. So, you can declare a new function as an alias to an existing function:

```
// A second pointer to the same block of code
int (*plus_two)(int) = add_two;
```

====================================================================

**Input and output**
By default, stdio.h uses input from “stdin,” send output to “stdout,” and errors to “stderr.” In order to use the following you must `#include <stdio.h>`. When printing, don’t forget to add \n if you need a new line in or after the output!

```
printf(p, ...)  // Write to stdout with format string p
scanf(p, ...)  // Read from stdin with format string p
getchar()  // Get a character from stdin
ungetchar(c)  // Put char c back into stdin to re-read
putchar(c)  // Write c into stdout
gets(s)  // Read stdin to s until EOF or \n
puts(s) // Write a string to stdout
fopen( fname, “r”)  // Open file fname, read permissions
fclose(f)  // Close files-always close after you are finished with file
fprintf(f, p, ...) // printf but to file f
fscanf(f, p, ...) // scanf but from file f
fgets(s, n, f)  // Read n-1 of f into s, or to EOF or \n
fputs(s, f)  // Write s to f
feof(f)  // Test EOF indicator
```

*NOTE: \n is newline and \o is the null terminator at the end of a string*
In order to send output to stderr, you must use fprintf(). Since unix systems essentially treat everything as a file, you can send output to stderr with

```
fprintf(stderr, “This is my %s string.\n”, “formatted error”);
// Sends “This is my formatted error.\n” to stderr
```

====================================================================

**Format strings and escape characters**
To use printf and similar functions, you typically define a format string. The format string is a template for your output, which contains format characters that will be filled in with the parameters passed to printf after the format string.

```
printf(“This format string prints %d, an int and %g, a float.”,
       5, 42.42);
```

Format characters are preceded by %, special characters are preceded by ‘\’ (the escape character)

| | | | |
|---|---|---|---|
| %c | character | %d | signed decimal integer |
| %s | string | %g | general float |
| %x | hex int | %p | pointer |
| %% | percent char | \n | new line |
| \0 | null term | \t | tab |
| \r | return | \\ | back-slash |

====================================================================

**Structures** (example structs.c)
Structs collect multiple fields into a single container. For example, tuples, data frames, or json dictionaries might be implemented via a struct.

```
// Create a new structure called name with fields x, y
struct name{ type x; type y; };

/* Define struct name and initialize var_name at the same time */
struct name{ type x; type y; } var_name;

// Initialize var_name as previously defined struct name
struct name var_name = {a, b};
var_name.x;  // Reference field x in var_name
var_name.y = 5;  // Assign to field y directly

// Declare bit field x with members a, b each four bits.
struct{char a:4; char b:4;} x;

// create struct and instantiate s
struct { int n; double root;} s;
/* Initialize s.n as 16 then initialize s.root as sqrt(16) */
s.root = sqrt((s.n = 16));
```

====================================================================

**Types** (example types.c)
You can create new types and alias existing types

```
typedef unsigned short uint16;  // Alias unsigned shot as uint16
uint16 x = 65535;  // Use new type as normal

// Define a new struct and create a new type from it
typedef struct struct_name{type a; type b;} type_name;

// Examples
typedef struct coord_struct{int x, y;} coord;
coord c_1 = { 15, 16 };  // Declare var c_1 as coord

// Create coordinate based on y = mx + b
coord make_coord( int m, int x, int b ){
   coord c;
   c.y = m*(c.x=5)+b;  // Note the assignment inside the operation
   return c;
}
```

=====================================================================

**Pointers** (example pointers.c)

Pointers hold an address in memory, such as the location of an item in a linked list or the location of an index in a string.

```
type *name;  // Pointer name to type
type *fxn();  // Function returning a pointer to type
void *name;  // Generic pointer
&name          // address of object name
type *name[2];  // An array of 2 pointers to type
type (*name)[2];  // A pointer to an array of 2 of type
```

For example, the following creates a pointer to a memory address called “int_array”, it then creates enough room in memory for 10 integers and sets the address to the first integer in that space.

```
int length = 10;
int *int_array; y = malloc( sizeof( int ) * length );
```

Pointers hold an address to a location in memory. Because it is an address, it does not have the same size as the data type it points to. The size of a pointer will always be the same as the number of bytes your system uses to address memory. For example, on a 32-bit system, memory addresses are 4 bytes, so pointers are 4 bytes.

```
sizeof(char) != sizeof(char *)  // because 1 byte != 4 bytes on
                                //  a 32-bit system.
```

Since pointers hold an address, adding 1 to any pointer computes the next address by using the size of its type.

```
int a = 10;
int *p = &a;
p = p + 1;
printf(“%s”, (p == (&a + sizeof(int))) ? “TRUE”:”FALSE” );  // TRUE
```

=====================================================================

**Memory**

You can directly address and manipulate the memory that is assigned to the program by the OS. If you reference memory not allocated to the program/process you will raise a `segmentation fault`.

```
malloc(x);  // Alloc x bytes return memory loc if success; NULL if fail
free(ptr);  // Release memory allocated to ptr
realloc(ptr, size);  // Resize memory assigned to ptr
calloc(n, size); // Allocate n blocks of size and fill with zeros.
```

Examples

```
// Allocate memory for a variable.
type *x; x = malloc( sizeof( type ) );
// Allocate memory for an array of length for type
type *y; y = malloc( sizeof( type ) * length );
// Allocate memory for an array of length for type and fill with zeros.
type *y; y = calloc(length, sizeof( type ));
```

=====================================================================

**Useful libraries.** Reference these for class/type utilities and constants such as `strlen()` and integer typed limits like `INT_MIN` and `INT_MAX`.

| Name | Description |
|---|---|
| ctype.h | Character class tests like islower() and isdigit() |
| float.h | Limits (see below) specific to floats and doubles |
| limits.h | Contains values for constants on a 32-bit Unix system, ex INT_MAX |
| math.h | Common math functions like sqrt(), floor(), and cos() |
| stdbool.h | Standard definition for Boolean type |
| stdint.h | Extended integer types with different sizes and value ranges |
| stdio.h | Standard utils for input and output like printf(), both console and files |
| stdlib.h | Standard utilities and functions like malloc(), qsort(), and randomicity |
| string.h | String utilities and functions like strncopy() |

=====================================================================

**ANSI Standard Libraries.** Come with default installation of C.

```
Assert.h       ctype.h        errno.h        float.h
Limits.h       locale.h       math.h         setjmp.h
signal.h       stdarg.h       stddef.h       stdlib.h
string.h       time.h
```

=====================================================================

**Comments and Style**

All lines should be less than 80 characters unless your organization says otherwise.

```
/* When wrapping to a new line, operators should follow their operands
and add clarity, almost like doing math on scratch paper.
*/
int a = x
        + y
        + z;

/* When wrapping parentheticals, break after the comma and line up
under the first parenthesis
*/
printf(“I’m a really long statement printing %d, %d, and %d.\n”,
      5, 4, 1);

// This comment’s length includes spaces and the “//”
// Comments start with a space and capital letter
statement;  // Clarify your code
statement;  // Don’t repeat the obvious
x = x + 1;  // Increment x (Don’t repeat the obvious in comments)
statement;  // Two spaces after the ;

/* Describe the program or function with multi-line
   comments */
char *string_copy(){
   /* Returns an identical copy of the given string but at a different
   address.

   Input: s - character array of unspecified length
   Output: t – a character array identical to s
   */
}
```

Variables should be named clearly and descriptively. Use camel-case (ex, thisIsAVariableName) or underscores (ex, this_is_a_variable_name) based on your organization's style guide.

===================================================================

**Compiling.** Compiling is usually done with gcc, the GNU Compiler Collection. Common commands from a Unix shell (including Windows Subsystem for Linux)

```
# Compile prog.c into prog; run via ./prog
gcc -o prog prog.c
# Compile and allow for debugging
gcc -g -o prog prog.c
```

Your system may require you to `chmod` to add execution to `prog` before it will allow you to execute via `./prog`

===================================================================

**Magic Numbers:**

Magic numbers can either represent file format indicators or can represent unnamed constants within your code.

Format indicators are typically numbers at the beginning of a file (like a .png or .jpg) allow the system to determine what type of file it is and perform error checking. The list of possible format indicators is long, and each has its own idiosyncrasies. You can research each more at https://en.wikipedia.org/wiki/List_of_file_signatures

Within a program, a magic number is an unnamed constant, typically appearing without prior definition or description. They often appear when programmers implement math or loop calculations that they understand without defining a constant or variable first. For example,

```
for (int i = 0; i < 42; i++){
   i = i / 2;
}
```

In this case, reading the for-loop we have no idea why the number 42 is used or when it will be used again. Instead we should write:

```
const int meaning_of_life = 42;
for (int i = 0; i < meaning_of_life; i++){
   i = i / 2;
}
```

Now, reading the code, we immediately know that we're performing an operation until we reach the value for the "meaning of life". (Though you'll notice that we'll never reach it...)

Your organization's style guide will help you to determine when to use constants, variables, or magic numbers with comments. In general, the more human readable the code, the better the code.

===================================================================

**References:**

University of Washington's C Cheatsheet
https://courses.cs.washington.edu/courses/cse351/14sp/sections/1/Cheatsheet-c.pdf

University of Texas's C Cheatsheet
https://users.ece.utexas.edu/~adnan/c-refcard.pdf

Ashlyn Black's C Cheatsheet
https://www.cheatography.com/ashlyn-black/cheat-sheets/c-reference/pdf/

Beginning C Programming for Dummies Cheatsheet
https://www.dummies.com/programming/c/beginning-c-programming-for-dummies-cheat-sheet/

String copy references
https://www.techiedelight.com/implement-strcpy-function-c/
https://www.systutorials.com/docs/linux/man/3p-strcpy/

Format Indicators
https://en.wikipedia.org/wiki/File_format#Magic_number
https://en.wikipedia.org/wiki/List_of_file_signatures

===================================================================

**Recommended Reading:**

- Learn C the Hard Way by Zed Shaw
- "Smashing the Stack for Fun and Profit" by Aleph One, Phrack 49, Volume 7, File 14