C Reference Cheat Sheet

by Ashlyn Black (Ashlyn Black) via cheatography.com/20410/cs/3196/

Number Literals

Integers

`0b11111111`	binary	`0B11111111`	binary
`0377`	octal	`255`	decimal
`0xff`	hexadecimal	`0xFF`	hexadecimal

Real Numbers

`88.0f` / `88.1234567f`

single precision float (`f` suffix)

`88.0` / `88.123456789012345`

double precision float (no `f` suffix)

Signage

`42` / `+42`	positive	`-42`	negative

Binary notation `0b...` / `0B...` is available on GCC and most but not all C compilers.

Variables

Declaring

`int x;`	A variable.
`char x = 'C';`	A variable & initialising it.
`float x, y, z;`	Multiple variables of the same type.

Variables (cont)

`const int x = 88;`	A constant variable: can't assign to after declaration (compiler enforced.)

Naming

`johnny5IsAlive;` ✔	Alphanumeric, not a keyword, begins with a letter.
~~`2001`~~`ASpaceOddysey;` ✖	Doesn't begin with a letter.
~~`while`~~`;` ✖	Reserved keyword.
`how`~~` exciting!`~~`;` ✖	Non-alphanumeric.

~~`iamave ryl ong var iab len ame ohm ygo shy esiam`~~`;` ✖

Longer than 31 characters (C89 & C90 only)

Constants are `CAPITALISED`. Function names usually take the form of a verb eg. `plotRobotUprising()`.

Primitive Variable Types

**applicable but not limited to most ARM, AVR, x86 & x64 installations*

`[class] [qualifier] [unsigned] type/void name;`

by ascending arithmetic conversion

Integers

Type	*Bytes*	*Value Range*

Primitive Variable Types (cont)

`char`	1	`unsigned` **OR** `signed`
`unsigned char`	1	0 to 2^8-1
`signed char`	1	-2^7 to 2^7-1
`int`	2 / 4	`unsigned` **OR** `signed`
`unsigned int`	2 / 4	0 to 2^{16}-1 **OR** 2^{31}-1
`signed int`	2 / 4	-2^{15} to 2^{15}-1 **OR** -2^{31} to 2^{32}-1
`short`	2	`unsigned` **OR** `signed`
`unsigned short`	2	0 to 2^{16}-1
`signed short`	2	-2^{15} to 2^{15}-1
`long`	4 / 8	`unsigned` **OR** `signed`
`unsigned long`	4 / 8	0 to 2^{32}-1 **OR** 2^{64}-1
`signed long`	4 / 8	-2^{31} to 2^{31}-1 **OR** -2^{63} to 2^{63}-1

Primitive Variable Types (cont)

`long long`	8	`unsigned` **OR** `signed`
`unsigned long long`	8	0 to 2^{64}-1
`signed long long`	8	-2^{63} to 2^{63}-1
Floats		
Type	*Bytes*	*Value Range (Normalized)*
`float`	4	$\pm1.2\times10^{-38}$ to $\pm3.4\times10^{38}$
`double`	8 / 4	$\pm2.3\times10^{-308}$ to $\pm1.7\times10^{308}$ **OR** alias to `float` for AVR.
`long double`	ARM: 8, AVR: 4, x86: 10, x64: 16	
Qualifiers		
`const type`	Flags variable as read-only (compiler can optimise.)	

Primitive Variable Types (cont)

`volatile type`	Flags variable as unpredictable (compiler cannot optimise.)
Storage Classes	
`register`	Quick access required. May be stored in RAM **OR** a register. Maximum size is register size.
`static`	Retained when out of scope. `static` global variables are confined to the scope of the compiled object file they were declared in.
`extern`	Variable is declared by another file.
Typecasting	
`(type)a`	Returns `a` as data `type`.

Primitive Variable Types (cont)

```
char x = 1, y = 2; float z = (float) x / y;
```

Some types (denoted with **OR**) are architecture dependant.

There is no primitive boolean type, only zero (false, `0`) and non-zero (true, usually `1`.)

Extended Variable Types

```
[class] [quali fier] type name;
```

by ascending arithmetic conversion

From the `stdint.h` Library

Type	*Bytes*	*Value Range*
`int8_t`	1	-2^7 to 2^7-1
`uint8_t`	1	0 to 2^8-1
`int16_t`	2	-2^{15} to $2^{15}-1$
`uint16_t`	2	0 to $2^{16}-1$
`int32_t`	4	-2^{31} to $2^{31}-1$
`uint32_t`	4	0 to $2^{32}-1$
`int64_t`	8	-2^{63} to $2^{63}-1$
`uint64_t`	8	0 to $2^{64}-1$

From the `stdbool.h` Library

Type	*Bytes*	*Value Range*

Extended Variable Types (cont)

`bool`	1	`true` / `false` or `0` / `1`

The `stdint.h` library was introduced in C99 to give integer types architecture-independent lengths.

Structures

Defining

`struct strctName{ type x; type y; };`	A structure type `strctName` with two members, `x` and `y`. *Note trailing semicolon*
`struct item{ struct item *next; };`	A structure with a recursive structure pointer inside. Useful for linked lists.

Declaring

`struct strctName varName;`	A variable `varName` as structure type `strctName`.
`struct strctName *ptrName;`	A `strctName` structure type pointer, `ptrName`.
`struct strctName{ type a; type b; } varName;`	Shorthand for defining `strctName` and declaring `varName` as that structure type.

Structures (cont)

`struct strctName varName = { a, b };`	A variable `varName` as structure type `strctName` and initialising its members.

Accessing

`varName.x`	Member `x` of structure `varName`.
`ptrName->x`	Value of structure pointer `ptrName` member `x`.

Bit Fields

`struct{char a:4, b:4} x;`	Declares `x` with two members `a` and `b`, both four bits in size (0 to 15.)

Array members can't be assigned bit fields.

Type Definitions

Defining

`typedef unsigned short uint16;`	Abbrevia a longer type nar to `uint`
`typedef struct structName{int a, b;}newType;`	Creating `ewType` from a structure

Type Definitions (cont)

`typedef enum typeName{false, true}bool;`	Creating an enumerated `bool` type.
Declaring	
`uint16 x = 65535;`	Variable `x` as type `uint16`.
`newType y = {0, 0};`	Structure `y` as type `newType`.

Unions

Defining	
`union uName{int x; char y[8];}`	A union type `uName` with two members, `x` & `y`. Size is same as biggest member size.
Declaring	
`union uN vName;`	A variable `vName` as union type `uN`.
Accessing	
`vName.y[int]`	Members cannot store values concurrently. Setting `y` will corrupt `x`.

Unions are used for storing multiple data types in the same area of memory.

Enumeration

Defining	
`enum bool { false, true };`	A custom data type `bool` with two possible states: `false` or `true`.
Declaring	
`enum bool varName;`	A variable `varName` of data type `bool`.
Assigning	
`varName = true;`	Variable `varName` can only be assigned values of either `false` or `true`.
Evaluating	
`if(varName == false)`	Testing the value of `varName`.

Pointers

Declaring	
`type *x;`	Pointers have a data `type` like normal variables.
`void *v;`	They can also have an incomplete type. Operators other than assignment cannot be applied as the length of the type is unknown.

Pointers (cont)

`struct type *y;`	A data structure pointer.
`type z[];`	An array/string name can be used as a pointer to the first array element.
Accessing	
`x`	A memory address.
`*x`	Value stored at that address.
`y->a`	Value stored in structure pointer `y` member `a`.
`&varName`	Memory address of normal variable `varName`.
`*(type *)v`	Dereferencing a `void` pointer as a `type` pointer.

A pointer is a variable that holds a memory location.

Arrays

Declaring

`type name[int];`	You set array length.
`type name[int] = {x, y, z};`	You set array length and initialise elements.

Arrays (cont)

`type name[int] = {x};`	You set array length and initialise all elements to `x`.
`type name[] = {x, y, z};`	Compiler sets array length based on initial elements.

Size cannot be changed after declaration.

Dimensions

`name[int]`	One dimension array.
`name[int][int]`	Two dimensional array.

Accessing

`name[int]`	Value of element `int` in array `name`.
`*(name + int)`	Same as `name[int]`.

Elements are contiguously numbered ascending from `0`.

`&name[int]`	Memory address of element `int` in array `name`.
`name + int`	Same as `&name[int]`.

Elements are stored in contiguous memory.

Measuring

`sizeof(array) / sizeof(arrayType)`	Returns length of `array`. *(Unsafe)*

Arrays (cont)

`sizeof(array) / sizeof(array[0])`	Returns length of `array`. *(Safe)*

Strings

`'A'` character	Single quotes.
`"AB"` string	Double quotes.
`\0`	Null terminator.

Strings are `char` *arrays.*

`char name[4] = "Ash";`

is equivalent to

`char name[4] = {'A', 's', 'h', '\0'};`

`int i; for(i = 0; name[i]; i++){}`

`\0` *evaluates as false.*

Strings must include a `char` element for `\0`.

Escape Characters

`\a`	alarm (bell/beep)	`\b`	backspace
`\f`	formfeed	`\n`	newline
`\r`	carriage return	`\t`	horizontal tab
`\v`	vertical tab	`\\`	backslash
`\'`	single quote	`\"`	double quote

Escape Characters (cont)

`\?`	question mark
`\nnn`	Any octal ANSI character code.
`\xhh`	Any hexadecimal ANSI character code.

Functions

Declaring

`type/void funcName([args...]){ [return var;] }`

Function names follow the same restrictions as variable names but must ***also*** *be unique.*

`type/void`	Return value type (`void` if none.)
`funcName()`	Function name and argument parenthesis.
`args...`	Argument types & names (`void` if none.)
`{}`	Function content delimiters.

Functions (cont)

`return var;`	Value to return to function call origin. Skip for `void` type functions. Functions exit immediately after a `return`.
By Value vs By Pointer	
`void f(type x); f(y);`	Passing variable `y` to function `f` argument `x` (by value.)
`void f(type *x); f(array);`	Passing an array/string to function `f` argument `x` (by pointer.)
`void f(type *x); f(structure);`	Passing a structure to function `f` argument `x` (by pointer.)
`void f(type *x); f(&y);`	Passing variable `y` to function `f` argument `x` (by pointer.)
`type f(){ return x; }`	Returning by value.
`type f(){ type x; return &x; }`	Returning a variable by pointer.

Functions (cont)

`type f(){ static type x[]; return &x; }`	Returning an array/string/str by pointer. The `tic` qualifier is necessary othe `x` won't exist a the function ex

Passing by pointer allows you to change the originating variable withi function.

Scope

`int f(){ int i = 0; }` ~~`i++;`~~✖

`i` is declared inside `f()`, it doesn't exist outside that function.

Prototyping

`type funcName(args...);`

Place before declaring or referencing respective function (usually befor n.)

`type funcName([args...])`	Same `type`, n and `args...` respective func
`;`	Semicolon inst function delimi

main()

```
int main(int argc, char *argv[]){return int;}
```

Anatomy

`int main`	Program entry point.
`int argc`	# of command line arguments.
`char *argv[]`	Command line arguments in an array of strings. #1 is always the program filename.
`return int;`	Exit status (`integer`) returned to the OS upon program exit.

Command Line Arguments

`app two 3`	Three arguments, "`app`", "`two`" and "`3`".
`app "two 3"`	Two arguments, "`app`" and "`two 3`".

`main` is the first function called when the program executes.

Conditional (Branching)

if, else if, else

`if(a) b;`	Evaluates `b` if `a` is true.
`if(a){ b; c; }`	Evaluates `b` and `c` if `a` is true.
`if(a){ b; }else{ c; }`	Evaluates `b` if `a` is true, `c` otherwise.

Conditional (Branching) (cont)

```
if(a){ b; }else if(c){ d; }else{ e; }
```

switch, case, break

```
switch(a){ case b: c; }
```

```
switch(a){ default: b; }
```

```
switch(a){ case b: case c: d; }
```

```
switch(a){ case b: c; case d: e; default: f; }
```

```
switch(a){ case b: c; break; case d: e; break; defau
```

Iterative (Looping)

while

`int x = 0; while(x < 10){ x += 2; }`

Loop skipped if test condition initially false.

`int x = 0;`	Declare and initialise integer `x`.
`while()`	Loop keyword and condition parenthesis.
`x < 10`	Test condition.
`{}`	Loop delimiters.
`x += 2;`	Loop contents.

do while

`char c = 'A'; do { c++; } while(c != 'Z');`

Always runs through loop at least once.

`char c = 'A';`	Declare and initialise character `c`.
`do`	Loop keyword.
`{}`	Loop delimiters.
`c++;`	Loop contents.
`while();`	Loop keyword and condition parenthesis. *Note semicolon.*
`c != 'Z'`	Test condition.

for

`int i; for(i = 0; n[i] != '\0'; i++){}` (C89)

Iterative (Looping) (cont)

OR

`for(int i = 0; n[i] != '\0'; i++){}` (C99+)

Compact increment/decrement based loop.

`int i;`	Declares integer `i`.
`for()`	Loop keyword.
`i = 0;`	Initialises integer `i`. *Semicolon.*
`n[i] != '\0';`	Test condition. *Semicolon.*
`i++`	Increments `i`. *No semicolon.*
`{}`	Loop delimiters.

continue

`int i=0; while(i<10){ i++; continue; i--;}`

Skips rest of loop contents and restarts at the beginning of the loop.

break

`int i=0; while(1){ if(x==10){break;} i++; }`

Skips rest of loop contents and exits loop.

Console Input/Output

`#include <stdio.h>`

Characters

`getchar()`	Returns a single character's ANSI code from the input stream buffer as an *integer*. *(safe)*
`putchar(int)`	Prints a single character from an ANSI code *integer* to the output stream buffer.

Strings

`gets(strName)`	Reads a line from the input stream into a string variable. *(Unsafe, removed in C11.)*
Alternative	
`fgets(strName, length, stdin);`	Reads a line from the input stream into a string variable. *(Safe)*
`puts("string")`	Prints a string to the output stream.

Formatted Data

Console Input/Output (cont)

`scanf("%d", &x)`	Read value/s (type defined by format string) into variable/s (type must match) from the input stream. Stops reading at the first whitespace. `&` *prefix not required for arrays (including strings.) (unsafe)*
`printf ("I love %c %d!", 'C', 99 )`	Prints data (formats defined by the format string) as a string to the output stream.
Alternative	

Console Input/Output (cont)

`fgets(strName, length, stdin); sscanf(strName, "%d", &x);`	Uses `fgets` to limit the input length, then uses `sscanf` to read the resulting string in place of `scanf` *(safe)*

The stream buffers must be flushed to reflect changes. String terminator characters can flush the output while newline characters can flush the input.

Safe functions are those that let you specify the length of the input. *Unsafe* functions do not, and carry the risk of memory overflow.

File Input/Output

```
#include <stdio.h>
```

Opening

```
FILE *fptr = fopen(filename, mode);
```

`FILE *fptr`	Declares `fptr` as a FILE type pointer (stores stream location instead of memory location.)
`fopen()`	Returns a stream location pointer if successful, `0` otherwise.

File Input/Output (cont)

`filename`	String containing file's directory path & name.
`mode`	String specifying the file access mode.

Modes

`"r"` / `"rb"`	Read existing text/binary file.
`"w"` / `"wb"`	Write new/over existing text/binary file.
`"a"` / `"ab"`	Write new/append to existing text/binary file.
`"r+"` / `"r+b"` / `"rb+"`	Read and write existing text/binary file.
`"w+"` / `"w+b"` / `"wb+"`	Read and write new/over existing text/binary file.
`"a+"` / `"a+b"` / `"ab+"`	Read and write new/append to existing text/binary file.

Closing

C Reference Cheat Sheet

by Ashlyn Black (Ashlyn Black) via cheatography.com/20410/cs/3196/

File Input/Output (cont)

`fclose(fptr);`	Flushes buffers and closes stream. Returns `0` if successful, `EOF` otherwise.
Random Access	
`ftell(fptr)`	Return current file position as a long integer.
`fseek(fptr, offset, origin);`	Sets current file position. Returns *false* is successful, *true* otherwise. The `offset` is a long integer type.
Origins	
`SEEK_SET`	Beginning of file.
`SEEK_CUR`	Current position in file.
`SEEK_END`	End of file.
Utilities	
`feof(fptr)`	Tests end-of-file indicator.
`rename(strOldName, strNewName)`	Renames a file.
`remove(strName)`	Deletes a file.
Characters	

File Input/Output (cont)

`fgetc(fptr)`	Returns character read or EOF if unsuccessful. *(safe)*
`fputc(int c, fptr)`	Returns character written or EOF if unsuccessful.
Strings	
`fgets(char *s, int n, fptr)`	Reads `n-1` characters from file `fptr` into string `s`. Stops at `EOF` and `\n`. *(safe)*
`fputs(char *s, fptr)`	Writes string `s` to file `fptr`. Returns non-negative on success, `EOF` otherwise.
Formatted Data	
`fscanf(fptr, format, [...])`	Same as `scanf` with additional file pointer parameter. *(unsafe)*
`fprintf(fptr, format, [...])`	Same as `printf` with additional file pointer parameter.
Alternative	

File Input/Output (cont)

`fgets(strName, length, fptr); sscanf(strName, "%d", &x);`	Uses `fgets` to limit the input length, then uses `sscanf` to read the resulting string in place of `scanf`. *(safe)*
Binary	
`fread(void *ptr, sizeof(element), number, fptr)`	Reads a `number` of `elements` from `fptr` to array `*ptr`. *(safe)*
`fwrite(void *ptr, sizeof(element), number, fptr)`	Writes a `number` of `elements` to file `fptr` from array `*ptr`.

Safe functions are those that let you specify the length of the input. *Unsafe* functions do not, and carry the risk of memory overflow.

Placeholder Types (f/printf And f/scanf)

`printf("%d%d...", arg1, arg2...);`

Type	Example	Description
`%d` or `%i`	`-42`	Signed decimal integer.

Placeholder Types (f/printf And f/scanf) (cont)

`%u`	`42`	Unsigned decimal integer.
`%o`	`52`	Unsigned octal integer.
`%x` or `%X`	`2a or 2A`	Unsigned hexadecimal integer.
`%f` or `%F`	`1.21`	Signed decimal float.
`%e` or `%E`	`1.21e+9` or `1.21E+9`	Signed decimal w/ scientific notation.
`%g` or `%G`	`1.21e+9` or `1.21E+9`	Shortest representation of `%f`/`%F` or `%e`/`%E`.
`%a` or `%A`	`0x1.207c8ap+30` or `0X1.207C8AP+30`	Signed hexadecimal float.
`%c`	`a`	A character.
`%s`	`A String.`	A character string.
`%p`		A pointer.
`%%`	`%`	A percent character.

Placeholder Types (f/printf And f/scanf) (cont)

`%n`	No output, saves # of characters printed so far. Respective printf argument must be an integer pointer.

The pointer format is architecture and implementation dependant.

Placeholder Formatting (f/printf And f/scanf)

`%[Flags][Width][.Precision][Length]Type`

Flags

`-`	Left justify instead of default right justify.
`+`	Sign for both positive numbers and negative.
`#`	Precede with `0`, `0x` or `0X` for `%o`, `%x` and `%X` tokens.
`space`	Left pad with spaces.
`0`	Left pad with zeroes.

Width

`integer`	Minimum number of characters to print: invokes padding if necessary. Will not truncate.
`*`	Width specified by a preceding argument in `printf`.

Placeholder Formatting (f/printf And f/scanf) (cont)

Precision

`.integer`	Minimum # of digits to print for `%d`, `%i`, `%o`, `%u`, `%x`, `%X`. Left pads with zeroes. Will not truncate. Skips values of 0.
	Minimum # of digits to print after decimal point for `%a`, `%A`, `%e`, `%E`, `%f`, `%F` (default of 6.)
	Minimum # of significant digits to print for `%g` & `%G`.
	Maximum # of characters to print from `%s` (a string.)
`.`	If no `integer` is given, default of 0.
`.*`	Precision specified by a preceding argument in `printf`.

Length

`hh`	Display a `char` as `int`.
`h`	Display a `short` as `int`.
`l`	Display a `long` integer.
`ll`	Display a `long long` integer.
`L`	Display a `long double` float.
`z`	Display a `size_t` integer.

C Reference Cheat Sheet

by Ashlyn Black (Ashlyn Black) via cheatography.com/20410/cs/3196/

Placeholder Formatting (f/printf And f/scanf) (cont)

`j`	Display a `intmax_t` integer.
`t`	Display a `ptrdiff_t` integer.

Preprocessor Directives

`#include <inbuilt.h>`	Replaces line with contents of a standard C header file.
`#include "./custom.h"`	Replaces line with contents of a custom header file. *Note dir path prefix & quotations.*
`#define NAME value`	Replaces all occurrences of `NAME` with `value`.

Comments

```
// We're single-line comments!
// Nothing compiled after // on these lines.
/* I'm a multi-line comment!
      Nothing compiled between
      these delimi ters. */
```

C Reserved Keywords

`_Alignas`	`break`	`float`	`signed`
`_Alignof`	`case`	`for`	`sizeof`
`_Atomic`	`char`	`goto`	`static`
`_Bool`	`const`	`if`	`struct`
`_Complex`	`continue`	`inline`	`switch`
`_Generic`	`default`	`int`	`typedef`
`_Imaginary`	`do`	`long`	`union`
`_Noreturn`	`double`	`register`	`unsigned`
`_Static_assert`	`else`	`restrict`	`void`
`_Thread_local`	`enum`	`return`	`volatile`
`auto`	`extern`	`short`	`while`
`_A-Z...`	`__...`		

C / POSIX Reserved Keywords

`E[0-9]...`	`E[A-Z]...`	`is[a-z]...`	`to[a-z]...`
`LC_[A-Z]...`	`SIG[A-Z]...`	`SIG_[A-Z]...`	`str[a-z]...`
`mem[a-z]...`	`wcs[a-z]...`	`..._t`	

GNU Reserved Names

Cheatography

C Reference Cheat Sheet

by Ashlyn Black (Ashlyn Black) via cheatography.com/20410/cs/3196/

Header Reserved Keywords

Name	*Reserved By Library*
`d_...`	`dirent.h`
`l_...`	`fcntl.h`
`F_...`	`fcntl.h`
`O_...`	`fcntl.h`
`S_...`	`fcntl.h`
`gr_...`	`grp.h`
`..._MAX`	`limits.h`
`pw_...`	`pwd.h`
`sa_...`	`signal.h`
`SA_...`	`signal.h`
`st_...`	`sys/stat.h`
`S_...`	`sys/stat.h`
`tms_...`	`sys/times.h`
`c_...`	`termios.h`
`V...`	`termios.h`
`I...`	`termios.h`
`O...`	`termios.h`
`TC...`	`termios.h`
`B[0-9]...`	`termios.h`

Header Reserved Keywords (cont)

GNU Reserved Names

Heap Space

`#include <stdlib.h>`

Allocating

`malloc();`	Re me loc su `NU` oth
`type *x; x = malloc(sizeof(type));`	Me a v
`type *y; y = malloc(sizeof(type) * length );`	Me an arr
`struct type *z; z = malloc(sizeof(struct type));`	Me a s

Deallocating

`free(ptrName);`	Re the all `pt`

Reallocating

Heap Space (cont)

`realloc(ptrName, size);`	Attempts to resize the memory block assigned to `ptrName`.

The memory addresses you see are from virtual memory the operating system assigns to the program; they are not physical addresses.

Referencing memory that isn't assigned to the program will produce an OS segmentation fault.

The Standard Library

`#include <stdlib.h>`

Randomicity

`rand()`	Returns a (predictable) random integer between 0 and RAND_MAX based on the randomiser seed.
`RAND_MAX`	The maximum value `rand()` can generate.
`srand(unsigned integer);`	Seeds the randomiser with a positive integer.
`(unsigned) time(NULL)`	Returns the computer's tick-tock value. Updates every second.

The Standard Library (cont)

Sorting

`qsort(array, length, sizeof(type),`

`qsort()`	Sort using the QuickSort a
`array`	Array/string name.
`length`	Length of the array/string.
`sizeof(type)`	Byte size of each element
`compFunc`	Comparison function nam

compFunc

`int compFunc( const void *a, const void b* ){ return`

`int compFunc()`	Function name unimporta
`const void *a, const void *b`	Argument names unimpo
`return( *(int *)a - *(int *)b);`	Negative result swaps `b` f result of `0` doesn't swap.

C's inbuilt randomiser is cryptographically insecure: DO NOT use it for security applications.

The Character Type Library

	`#include <ctype.h>`
`tolower(char)`	Lowercase `char`.
`toupper(char)`	Uppercase `char`.
`isalpha(char)`	True if `char` is a letter of the alphabet, false otherwise.
`islower(char)`	True if `char` is a lowercase letter of the alphabet, false otherwise.
`isupper(char)`	True if `char` is an uppercase letter of the alphabet, false otherwise.
`isnumber(char)`	True if `char` is numerical (`0` to `9`) and false otherwise.
`isblank`	True if `char` is a whitespace character (`' '`, `'\t'`, `'\n'`) and false otherwise.

The String Library

	`#include <string.h>`
`strlen(a)`	Returns # of `char` in string `a` as an integer. Excludes `\0`. *(unsafe)*
`strcpy(a, b)`	Copies strings. Copies string `b` over string `a` up to and including `\0`. *(unsafe)*
`strcat(a, b)`	Concatenates strings. Copies string `b` over string `a` up to and including `\0`, starting at the position of `\0` in string `a`. *(unsafe)*
`strcmp(a, b)`	Compares strings. Returns *false* if string `a` equals string `b`, *true* otherwise. Ignores characters after `\0`. *(unsafe)*
`strstr(a, b)`	Searches for string `b` inside string `a`. Returns a pointer if successful, `NULL` otherwise. *(unsafe)*
Alternatives	
`strncpy(a, b, n)`	Copies strings. Copies `n` characters from string `b` over string `a` up to and including `\0`. *(safe)*

The String Library (cont)

`strncat(a, b, n)`	Concatenates strings. Copies `n` characters from string `b` over string `a` up to and including `\0`, starting at the position of `\0` in string `a`. *(safe)*
`strncmp(a, b, n)`	Compares first `n` characters of two strings. Returns *false* if string `a` equals string `b`, *true* otherwise. Ignores characters after `\0`. *(safe)*

Safe functions are those that let you specify the length of the input. *Unsafe* functions do not, and carry the risk of memory overflow.

The Time Library

`#include <time.h>`

Variable Types

`time_t`	Stores the calendar time.
`struct tm *x;`	Stores a time & date breakdown.
tm structure members:	
`int tm_sec`	Seconds, 0 to 59.
`int tm_min`	Minutes, 0 to 59.
`int tm_hour`	Hours, 0 to 23.
`int tm_mday`	Day of the month, 1 to 31.

The Time Library (cont)

`int tm_mon`	Month, 0 to 11.
`int tm_year`	Years since 1900.
`int tm_wday`	Day of the week, 0 to 6.
`int tm_yday`	Day of the year, 0 to 365.
`int tm_isdst`	Daylight saving time.
Functions	
`time(NULL)`	Returns unix epoch time (seconds since 1/Jan/1970.)
`time(&time_t);`	Stores the current time in a `time_t` variable.
`ctime(&time_t)`	Returns a `time_t` variable as a string.
`x = localtime( &time_t);`	Breaks `time_t` down into `struct tm` members.

Unary Operators

by descending evaluation precedence

`+a`	Sum of `0` (zero) and `a`. (0 + a)
`-a`	Difference of `0` (zero) and `a`. (0 - a)
`!a`	Complement (logical NOT) of `a`. (~a)

Unary Operators (cont)

`~a`	Binary ones complement (bitwise NOT) of `a`. (~a)
`++a`	Increment of `a` by `1`. (a = a + 1)
`--a`	Decrement of `a` by `1`. (a = a - 1)
`a++`	Returns `a` then increments `a` by `1`. (a = a + 1)
`a--`	Returns `a` then decrements `a` by `1`. (a = a - 1)
`(type)a`	Typecasts `a` as `type`.
`&a;`	Memory location of `a`.
`sizeof(a)`	Memory size of `a` (or `type`) in bytes.

Binary Operators

by descending evaluation precedence

`a * b;`	Product of `a` and `b`. (a × b)
`a / b;`	Quotient of dividend `a` and divisor `b`. Ensure divisor is non-zero. (a ÷ b)
`a % b;`	Remainder of *integers* dividend `a` and divisor `b`.
`a + b;`	Sum of `a` and `b`.
`a - b;`	Difference of `a` and `b`.

Binary Operators (cont)

`a << b;`	Left bitwise shift of `a` by `b` places. ($a \times 2^b$)
`a >> b;`	Right bitwise shift of `a` by `b` places. ($a \times 2^{-b}$)
`a < b;`	Less than. True if `a` is less than `b` and false otherwise.
`a <= b;`	Less than or equal to. True if `a` is less than or equal to `b` and false otherwise. (a ≤ b)
`a > b;`	Greater than. True if `a` is greater than than `b` and false otherwise.
`a >= b;`	Greater than or equal to. True if `a` is greater than or equal to `b` and false otherwise. (a ≥ b)
`a == b;`	Equality. True if `a` is equal to `b` and false otherwise. (a ⇔ b)
`a != b;`	Inequality. True if `a` is not equal to `b` and false otherwise. (a ≠ b)
`a & b;`	Bitwise AND of `a` and `b`. (a ∩ b)
`a ^ b;`	Bitwise exclusive-OR of `a` and `b`. (a ⊕ b)

Binary Operators (cont)

`a	b;`	Bitwise inclusive-OR of `a` and `b`. (a ∪ b)
`a && b;`	Logical AND. True if both `a` and `b` are non-zero. (Logical AND) (a ∩ b)
`a || b;`	Logical OR. True if either `a` or `b` are non-zero. (Logical OR) (a ∪ b)

Ternary & Assignment Operators

by descending evaluation precedence

`x ? a : b;`	Evaluates `a` if `x` evaluates as true or `b` otherwise. (if(x){ a; } else { b; })
`x = a;`	Assigns value of `a` to `x`.
`a *= b;`	Assigns product of `a` and `b` to `a`. (a = a × b)
`a /= b;`	Assigns quotient of dividend `a` and divisor `b` to `a`. (a = a ÷ b)
`a %= b;`	Assigns remainder of *integers* dividend `a` and divisor `b` to `a`. (a = a mod b)
`a += b;`	Assigns sum of `a` and `b` to `a`. (a = a + b)

Ternary & Assignment Operators (cont)

`a -= b;`	Assigns difference of `a` and `b` to `a`. (a = a - b)
`a <<= b;`	Assigns left bitwise shift of `a` by `b` places to `a`. ($a = a \times 2^{b}$)
`a >>= b;`	Assigns right bitwise shift of `a` by `b` places to `a`. ($a = a \times 2^{-b}$)
`a &= b;`	Assigns bitwise AND of `a` and `b` to `a`. (a = a ∩ b)
`a ^= b;`	Assigns bitwise exclusive-OR of `a` and `b` to `a`. (a = a ⊕ b)
`a |= b;`	Assigns bitwise inclusive-OR of `a` and `b` to `a`. (a = a ∪ b)

C Cheatsheet by Ashlyn Black

ashlynblack.com