Cheatsheet - (Moritz Makowski)

Get a **full reference guide** at tutorialspoint.

C was initially used for system development work, particularly the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as the code written in assembly language. Some examples of the use of C are:

- Operating Systems
- Language Compilers
- Language Interpreters
- Assemblers
- Databases
- Network Drivers
- Print Spoolers
- Modern Programs
- Text Editors
- Utilities

Compilation

File `hello.c` :

```c
#include <stdio.h>

int main() {
   printf("Hello, World! \n");
   return 0;
}
```

Compile:

```
# The name will probably be 'a.out'
$ gcc hello.c

# Specify a name for the executable (here: 'app_exec')
$ gcc hello.c -o app_exec
```

Execute:

```
$ ./a.out
Hello, World!

# or

$ ./app_exec
Hello, World!
```

Basics

Comments:

```
// Single Line Comment

/*
Multi-Line Comment
*/
```

Declaring Variables:

```
int myNum;
myNum = 15;

int myNum = 15;

int x, y;
x = 10;
y = 15

int x = 10, y = 15, z;
```

Data Types

Integers:

```
// 1 byte, signed, [-128, 127]
signed char num = 1;

// 1 byte, unsigned, [0, 255]
unsigned char num = 1;



```

```
// 2 byte, signed, [-32.768, 32.767]
short num = 1;

// 2 byte, unsigned, [0, 65.535]
unsigned short num = 1;



// 2 or 4 byte, signed, [-32.768, 32.767] or [-2.147.483.648, 2.147.483.647]
int num = 1;

// 2 or 4 byte, unsigned, [0, 65.535] or [0, 4.294.967.295]
unsigned int num = 1;



// 8 bytes, signed, [-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]
long num = 1;

// 8 bytes, unsigned, [0, 18.446.744.073.709.551.615]
unsigned long num = 1;
```

The size of an int (2 or 4 byte) can vary for each implementation of the compiler/each operating system.

If you want an integer size to be exact in every environment you can use the standardlibrary `stdint.h` :

```
#include <stdint.h>

int8_t num;
uint8_t num;

int16_t num;
uint16_t num;

int32_t num;
uint32_t num;

int64_t num;
uint64_t num;
```

Floating Point Numbers:

All of these fulfill the `IEEE 754` Standard for Floating Point Numbers.

```c
// 4 byte, single precision, [1.2E-38, 3.4E+38]
// Precision to 6 decimal places
float num = 1.0;

// 8 byte, double precision, [2.3E-308, 1.7E+308]
// Precision to 15 decimal places
double num = 1.0;

// 10 byte, [3.4E-4932, 1.1E+4932]
// Precision to 19 decimal places
long double num = 1.0;
```

Boolean:

There is no standard data type for booleans in C, but you can use the standard library `stdbool.h`

```c
#include <stdbool.h>

// true
Bool bool1 = true;
Bool bool2 = 1;

// false
Bool bool3 = false;
Bool bool4 = 2;
```

Characters:

```c
// 8-bit Ascii character, [0, 255], is stored as a number
char character = 'A';
```

Type Casting:

You can convert the value of an expression from one type to another using:

```
(type_name) expression;
```

Example:

```c
int sum = 17, count = 5;
double mean = sum / count;
// mean is now 3.0

int sum = 17, count = 5;
double mean = ((double) sum) / count;
// mean is now 3.4
```

Constants:

You can define constants in the beginning of a program as a preprocessor ...

```c
#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
```

... or inside the program ...

```c
const int  LENGTH = 10;
const int  WIDTH = 5;
const char NEWLINE = '\n';
```

... and use them later without being able to change them:

```c
int area = LENGTH * WIDTH;
```

Arrays

Regular Array:

In C you have to define the length of the array at the time of declaration. An array can only contain one type of variable.

```c
// Declaring
double balance[5];
double balance[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
double balance[5] = {0}; // All elements in array are 0

// Indexing elements
balance[3]; // = 4.0
```

Multidimensional Array:

```c
// Declaring
float mat[10][20]; // 10 rows, 20 columns
float mat[10][20] = {0}; // All elements in array are 0

// Indexing
mat[2][4]; // element in 3rd row and 5th column
```

Input/Output Methods

There are different print command available in C:

Format Specifiers:

- `%d` → Integer
- `%f` → Float
- `%c` → Char
- `%s` → String (Char Array)

Scanner and Printer:

```c
#include <stdio.h>

int main() {
    printf("Please enter: ");
    char input[8];
    scanf("%7s", input);  // The 7 limits the input characters so
                // that there will be no memory overflow
    printf("%s", input);
    return 0;
}
```

Loops and Conditionals

For Loop:

```c
for ( init; condition; increment ) {
   statement(s);
}
```

You don't have to include a condition or an increment though.

While Loop:

```
while(condition) {
   statement(s);
}
```

Do-While Statement:

```
do {
   statement(s);
} while( condition );
```

Break & Continue:

The `break` statement can also be used to jump out of a loop (Only the most inner loop).

```
break;
```

The `continue` statement stops this iteration of the loop and jumps to the next one.

```
continue;
```

Go To Statement:

You can set up jump-to statement to jump without having to include loops for example:

```
goto label;
..
.
label: statement;
```

However it is widely considered bad-practice because it makes the code certainly more complex and unreadable.

Switch Statement:

```
switch(expression) {

  case const_exp_1:
        statement(s);
        break; /* optional */

  case const_exp_2 :
```

```
    statement(s);
    break; /* optional */

  default : /* optional */
    statement(s);

}

```

If Statement:

```
if (condition1) {
  ...
} else if (condition2) {
  ...
} else {
  ...
}
```

The `__?__:__` Operator:

You can use the short-form of an if-else-statement:

```
Bool val = Exp1 ? Exp2 : Exp3;
```

If `Exp1` returns true than the `val` is equal to the value of `Exp2` else `val` is equal to the value of `Exp3`.

Math

Many mathematical operations (e.g. trigonometric functions) are only accessible by including the library `math.h`. Example functions are:

```
#include <math.h>

...

a = ceil(2.5);    // a = 3.0
b = floor(2.5);   // b = 2.0

c = fabs(120);    // c = 120.0
d = fabs(-120);   // d = 120.0

e = log(2.7);     // e = 0.993252 (logarithmus naturalis)
```

```
f = log10(10000); // f = 4.0

float PI = 3.14159265;
float RAD_PER_DEGREE = PI/180.0;
g = sin(RAD_PER_DEGREE * 30)      // Sinus of 30 degrees: g = 0.5
```

`math.h` contains many more functions which you can look up here.

Methods

```
return_type function_name( parameter list ) {
   body of the function
}
```

Example:

```
int max(int num1, int num2) {
  ...
  return result;
}
```

Structs

A `Struct` is a nice way of storing multiple related variables inside one enclosure.

There are two ways to define a `Struct`.

```
struct point_1 {
  float x;
  float y;
  float z;
};

struct {
  float x;
  float y;
  float z;
}  point_2;
```

You can use `point_1` to create many objects, because the first definition behaves like a *blueprint*. Initialize an instance of that struct with:

```
struct point_1 a;
a.x = 0.5;
a.y = 1;
a.z = 2;

struct point_1 b = {
  .x = 0.5,
  .y = 1,
  .z = 2
};
```

The second definition however will create an anonymous struct definiton which is only used once, to create the instance `point_2`. You can always change an instances contents:

```
point_2.x = 0.5;
point_2.y = 1;
point_2.z = 2;
```

Enums

`Enums` are used to store a sequence of values in one variable. Example: You have three states of a traffic light (red, yellow and green) and want to use integer keys to identify them instead of Strings like `"red"`, `"yellow"` and `"green"`.

```
enum color {
  RED,                  //  0
  YELLOW,               //  1
  GREEN = YELLOW + 2,   //  3
  BLUE,                 //  4
  GRAY = 17,            // 17
  BROWN                 // 18
};

enum color c = BLUE;

c == 4;  // true
```

The `Enum` automatically counts upwards but you can modify the counting.

Unions

A Union is a data structure that can have members of different types which share the same space in memory. The size of a union is the size of its largest component. The major benefit is, that you can write and read from and to the memory space in different ways and create complex operations.

```
union mix {
  int i;                      // 4 bytes
  struct {short lo, hi;} s;   // 4 bytes
  char c[4];                  // 4 bytes
} m;

// define as integer i
m.i = 0xFF00F00F;  // 1111 1111   0000 0000   11110000   0000 1111

// interpret as struct s
m.s.lo;            // 1111 1111   0000 0000
m.s.hi;            //                         11110000   0000 1111

// interpret as char array c
m.c[0];            // 1111 1111
m.c[1];            //             0000 0000
m.c[2];            //                         11110000
m.c[3];            //                                    0000 1111
```

Pointers

A pointer is a memory address to a location in memory which stores the corresponding value of a variable. So each variable has its `value` and its `pointer` (to the memory location).

Getting the address corresponding to a value is called *referencing*.

Getting the value corresponding to an address in memory is called *dereferencing*.

```
int value = 1264;

// Getting the pointer to 'value' = referencing the value
int *pointer = &value;

// Getting the value to 'pointer' = dereferencing the pointer
printf("%d", *pointer);
```

Initialize a `NULL` pointer with:

```c
int *pointer = 0;
```

`NULL` is an invalid memory address so the pointer does not point anywhere.

Passing Values to Functions

You can simply *pass data by value*:

```c
int add_10(int value) {
  return value + 10;
}
```

The function creates a copy of the passed variable and only manipulates that copy. The variable passed to the function remains unchanged.

The other way is to *pass data by reference*:

```c
void add_10(int *value) {
  *value += 10;
}
```

The function manipulates the variable passed to the function directly because the local copy of a pointer still references the same space in memory.

Arrays and Pointers

An array variable is in fact a pointer to the first element of an array.

```c
char string_buffer[1000];
char *pointer = string_buffer;

string_buffer[2] == *(pointer + 2);
```

When adding an `integer n` to a pointer, the memory address increases by the amount of memory space that is occupied by `n` units (`int`/`char`/...) of the variable type, that is stored in that memory location.

The difference is that *pointers are mutable* (you can store some other address in them) and *arrays are immutable* (cannot be reinitialized with another array).

Pointers and Structs

Trivial access to struct members:

```
struct my_struct a;
a.member = 5;
```

However you can modify the member of a struct with a pointer to that struct instance without dereferencing the pointer:

```
struct my_struct a;
(&a)->member = 5;
```

Memory Allocation

In many cases you don't know in advance how much memory space your program will need. However you can also allocated memory space at runtime. This is called *dynamic memory allocation*.

Use `malloc` from the `sdtlib.h` library for that:

```
#include <stdlib.h>

...

int *my_array = malloc(100 * sizeof(int));
```

The function returns a pointer to the beginning of the memory block.

However the memory space is not initialized to 0 so there will be random entries in that space. `calloc` does the same as `malloc` but also initializes the memory to 0:

```
int *my_array = calloc(100 * sizeof(int));
```

By using `free` you can deallocate a block of memory so that it can be reused for other tasks. If you don't do that *memory leaks* will occur.

```
free(my_array);
```