Cheatography

C Programming Cheat Sheet

by aaa via cheatography.com/27686/cs/8089/

Data types and Variables

Arrays

`<type>`	`<array_name>[<index>] = expression`	

Pointers

`<type>`	`*p`	Declares *p* a pointer.
`p =`	`&var`	Declares pointer to the variable *var*.
`<type>`	`**pp`	Declares *pp* a pointer to a pointer.
`pp =`	`&p`	Declares pointer to the pointer *p*.

Structures

`struct tag_name { <type> <element1>; <type> <element2>; }`	Structures allow a programmer to have a collection of elements of different types representing something.

Formatting

Escape Characters

`\a`	Alert bell
`\b`	Backspace
`\n`	Newline
`*\*`	Backslash
`\"`	Double quote
`\?`	Question Mark

Conversion Specifiers

`%c`	char
`%s`	string
`%d`	int
`%u`	unsigned int
`%ld`	long int
`%o`	octal
`%x`	hexadecimal

Formatting (cont)

`%d`	double

Formatted I/O

`%5.2f`	Width of the printed field. ie. `'123.5'` becomes `'  123.50'`.
`%04d`	Fills unused space with zeros. ie. `21` becomes `0021`.
`%-f`	Aligns the output to the right.
`% [aeiou]`	Remove all characters but vowels.
`% [^aeiou]`	Remove all the vowels.
`%d*%d*%d`	Eliminate unnecessary characters. ie. `1/1/2001` can be stored in three integers as `1`,`1` and `2001`.

Example:

```
int integer = 1;
printf("This is an integer: %d", integer);
```

Dynamic memory allocation

`void* malloc(int size)`	Allocates *size* contiguous bytes of memory and returns a void pointer to the first byte allocated.
`void* calloc(int items, int size)`	Allocates *iitems x size* contiguous **cleared** (set to 0) bytes of memory and returns a void pointer to the first byte allocated.
`void* calloc(void* ptr, int new_size)`	Resizes allocated memory being pointed at by *ptr* to be *size* bytes and returns a void pointer to the first byte allocated.
`void free(void* ptr)`	Frees memory that is pointed at by *ptr*.

Linked lists

`struct node { int x; struct node* next; }; ... struct node* head;`	A structure pointing to other nodes. The first element is assigned to a pointer *head*.
`struct node* ptr = head; while(ptr!= NULL){ printf("%d\n", ptr->x); ptr = ptr->next; }`	Traversing the list.
`while(head != NULL){ ptr = head->next;{ free(head);{ head = ptr;{ }`	Deleting an element. ! You need to free the elements in the right order.

Linked lists (cont)

`if(ptr == NULL){ ptr = malloc(sizeof(struct node)); ptr->x = 4; ptr->next = NULL; head = ptr; }`	Adding an element. ! You need to cycle to the end first.

A linked list is a dynamic data structure consisting of a**sequence of records** where each element contains a **link to the next record**. They can be linked **singularly**, **doubly** or **circularly**.

- ☑ Every node has a payloead and a link to the next node.
- ☑ The end is indicated by a*NULL* pointer.
- ☑ It needs a pointer to the first item in the list.

Basics of C

General functions

`char`	`getchar()`	Obtains character from input stream
`int`	`sizeof(void var)`	Returns size in bytes of

Mathematical functions

`double`	`sqrt(double x)`	Square root of x
`double`	`pow(double x, double y)`	x raised to the power of y
`double`	`abs(double x)`	Absolute value of x
`double`	`ceil(double x)`	Rounds x to the smallest int no less than x
`double`	`floor(double x)`	Rounds x to the largest int not greater than x

Command line Arguments

```
int main (int argc, char* argv[]){
    /*code*/
}
```


```
int main (int argc, char* argv[]){
    /*code*/
}
```

Pre-processing

Pre-processor identifiers	
`_LINE_`	Current line being compiled.
`_FILE_`	Name of source file.
`_DATE_`	Date of compilation (mm dd yy).
`_TIME_`	Time of compilation (hh:mm:ss)

Macros	
`#include <some_lib.h>`	The contents of *`#include`* are read and merged into the file.
`#define VAR VALUE`	Define a variable.

`#ifdef DEBUG`
`  expression`
`#endif`
	Define a variable.

`#ifdef DEBUG`
`  expression`
`#endif`
	Conditional compilation can be turned on by both setting *`#define DEBUG 1`* or by `-D` in the command line.

`#ifdef condition`
`  #error "Error message"`
`#endif`
	Prints text as error message.

🎓 Function-like macros are **pre-processed** and have **no type checking** and are **not checked for compilation errors**, but are **executed faster** than normal C functions.

Strings

`int`	**`printf`**`(char out)`	Prints **formatted** output to stdout
`int`	**`scanf`**`(char *input)`	Reads **formatted** input from stdin
`int`	**`puts`**`(char *input)`	Writes a string to stdout up to but **not including the null character**. A **newline character is appended** to the output.

Strings (cont)

`char*`	**`fgets`**`(char *str, int n, FILE *stream)`	Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either **(n-1)** characters are read, the **newline character** is read, or the **end-of-file** is reached, whichever comes first.
`char*`	**`strcpy`**`(char dest, char src)`	Pass a string to another string variable.
`int`	**`strcat`**`(char *dest, char *src)`	Appends the string pointed to by src to the end of the string pointed to by dest
`char*`	**`strlen(const char *str)`**	Computes the length of the string str up to, but not including the terminating null character.

Sockets	
`int socket(int domain, int type, int protocol)`	Creates a socket.
`int close(int sockid)`	Closes a socket.
`int bind(int sockid, struct sockaddr* addr, int addrlen)`	Selects the port which is going to be used and reserves it for use by the socket. **It can be skipped for TCP and UDP sockets**.
`int listen(int sockid, int backlog)`	Listens for connections. **It's only used by a TCP server**.
`int accept(int sockid, struct sockaddr* clientAddr, int* addrlen)`	Establishes a connection for a **TCP server**. ! addrlen should be set to `sizeof(clientAddr)`.
`int connect(int sockid, struct sockaddr* serverAddr, int addrlen)`	Establishes a connection for a **TCP client**. ! addrlen should be set to `sizeof(clientAddr)`.
`int send(int sockid, void* msg, int len, int flags)`	Sends a message to a **TCP client/server** with length `len`.
`int recv(int sockid, void* buffer, int len, int flags)`	Receives a message from a **TCP client/server** with length `len`.
`int sendto(int sockid, void* msg, int len, int flags, struct sockaddr* foreign, int addrlen)`	Sends a message to a **UDP client/server** with length `len`.
`int recvfrom(int sockid, void* msg, int len, int flags, struct sockaddr* foreign, int addrlen)`	Receives a message from a **UDPclient/server** with length `len`.