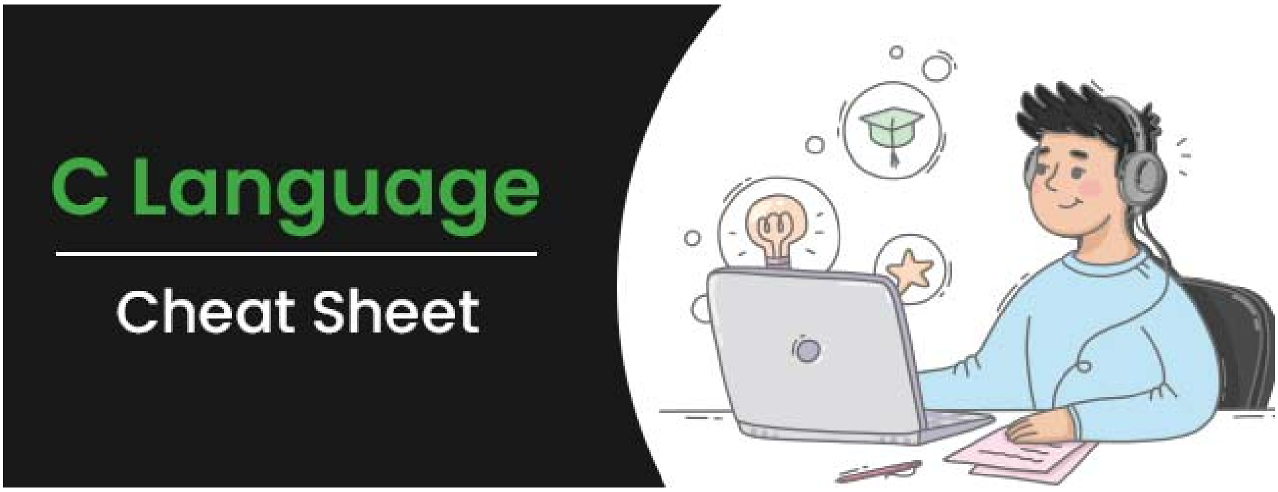
C Language - Cheat Sheet

tutorialspoint.com/cprogramming/c_language_cheatsheet.htm

This C language cheat sheet gives a quick overview of C language concepts starting from the basics to the advanced level. This cheat sheet is very useful for students, developers, and those who are preparing for an interview. Go through this cheat sheet to learn all basic and advanced concepts of C programming language.



Basis Structure of C Program

The basic structure of a C program gives you an idea about the basic statements that you need to use to write a program in C language. The following is the basic structure of a C program −

```
// Preprocessor directive/header file inclusion section
#include <stdio.h>

// Global declaration section

// the main() function
int main() {
   // Variable declarations section
   int x, y;

   // other code statements section

   // Return o
   return 0;
}
// Other user-defined function definition section
```

#include <stdio.h>

#include is a preprocessor directive that includes the header file in the C program. The **stdio.h** is a header file where all input and output-related functions are defined.

main() Function

The main() function is an entry point of a C program, the program's executions start from the main() function.

The below is the syntax of the main() function −

```
int main() {
   return 0;
}
```

Comments

There are two types of comments in C language. Single-line and multi-line comments. Comments are ignored by the compilers.

Single-line Comments

Use // to write a single-line comment.

```
// This is a single-line comment
```

Multi-line Comments

Use /* and */ before and after the text to write multi-line comments in C language.

```
/*
This is line 1
This is line 2
..
*/
```

Printing (printf() Function)

The printf() function is a library function to print the formatted text on the console output. Whenever you want to print anything, use the printf().

Example

```
printf("Hello world");
```

User Input (scanf() Function)

The scanf() function is used to take various types of inputs from the user.

Here is the syntax of the scanf() function −

```
scanf("format_specifier", &variable_name);
```

Format Specifiers

The following is the list of C format specifiers that are used in **printf()** and **scanf()** functions to print/input specific type of values.

Format Specifier	Type
%c	Character
%d	Signed integer
%e or %E	Scientific notation of floats
%f	Float values
%g or %G	Similar as %e or %E
%hi	Signed integer (short)
%hu	Unsigned Integer (short)
%i	Unsigned integer
%l or %ld or %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long
%lli or %lld	Long long
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x or %X	Hexadecimal representation

Example

```
#include <stdio.h>

int main(){

   int age = 18;

   float percent = 67.75;

   printf("Age: %d \nPercent: %f", age, percent);

   return 0;
}
```

Output

```
Age: 18
Percent: 67.750000
```

Data Types

The data types specify the type and size of the data to be stored in a variable. Data types are categorized in 3 sections −

- Basic Data Types
- Derived Data Types
- User-defined Data Types

Basic Data Types

The basic data types are the built-in data types in C language and they are also used to create derived data types.

Data Type	Name	Description
int	Integer	Represents integer Value
char	Character	Represents a single character
float	Float	Represents float value

Derived Data Types

The derived data types are derived from the basic data types. The derived data types are −

- Array
- Pointer

User-defined Data Types

The user-defined data types are created by the programmer to handle data of different type and based on the requirements. The user-defined data types are −

- Structures
- Unions
- Enumerations

Basic Input & Output

For basic input and output in C language, we use **printf()** and **scanf()** functions.

The **printf()** function is used to print the formatted text on the console.

```
printf("Hello world");
```

The **scanf()** function is used to take input from the user.

```
scanf("%d", &x); // Integer input
scanf("%f", &y); // float input
scanf("%c", &z); // Character Input
scanf("%s", name); // String input
```

Example of Basic Input and Output

```
#include <stdio.h>

int main() {
   int num;

   printf("Input any integer number: ");
   scanf("%d", &num);

   printf("The input is: %d\n", num);

   return 0;
}
```

Output

```
Input any integer number: The input is: 0
```

Identifiers

C identifiers are user-defined names for variables, constants, functions, etc. The following are the rules for defining identifiers −

- Keywords can't be used as identifiers.
- Only alphabets, underscore symbol (_), and digits are allowed in the identifier.
- The identifier must start either with an alphabet or an underscore.
- The same identifier can't be used as the name of two entities.
- Identifiers should be meaningful and descriptive.

Examples of Valid Identifiers

```
age, _name, person1, roll_no
```

Keywords

C keywords are the reversed words in the C compiler, which are used for specific purposes and must not be used as an identifier.

The following are the keywords in C language −

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

Variables

C variables are the name given to a storage area that our programs can use to access and manipulate the data.

Syntax of Declaring a Variable

```
data_type variable_name;
```

Escape Sequences

Escape sequences are the special characters followed by the escape (backward slash \). Escape sequences have special meanings and are used for printing those characters that cannot be printed normally.

Here is the list of escape sequences in C language −

Escape sequence	Meaning
\\	\ character
\'	' character

\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Operators

Operators are the special symbols that are used to perform specific mathematical or logical operations.

Below are the operators used in C language −

Operators	Symbols	Description
Assignment Operators	=, +=, -=, <<=	Performs assignment operations i.e., assigning values to variables.
Arithmetic Operators	+, -, *, /, %	Performs arithmetic operations.
Relational Operators	<, >, <=, >=, ==, !=	Performs comparisons on two operands.
Logical Operators	&&, ||, !	Performs logical operations such as logical AND, OR, and NOT.
Bitwise Operators	&, ^, |, <<, >>, ~	Performs bitwise operations.
Ternary Operator	? :	Performs conditional operation for decision-making.
Miscellaneous Operators	, sizeof, &, *, ⇒, .	Used for performing various other operations.

Example of Operators

```
result = num1 + num2;
if(result >=10){
   printf("Greater than 10.");
}
```

Conditional Statements

C language provides the following conditional statements −

- if Statement
- if-else Statement
- if-else-if Statement
- Nested if-else Statement
- Switch Statement
- Ternary Operator

if Statement

An if statement consists of a Boolean expression followed by one or more statements.

The syntax of if statement is −

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
}
```

if-else statement

An if-else statement can be followed by an optional else statement, which executes when the Boolean expression is false.

The syntax of the if statement is −

```
if (Boolean expr){
   Expression;
   . . .
}
else{
   Expression;
   . . .
}
```

if-else-if Statement

The if-else-if statement is also known as the ladder if-else. It is used to check multiple conditions when a condition is not true.

The syntax of if-else-if statement −

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

Nested if Statements

By using the nested if statements, you can use one if or else-if statement inside another if or else-if statement(s).

The syntax of nested if statements −

```
if (expr1){
   if (expr2){
      block to be executed when
      expr1 and expr2 are true
   }
   else{
      block to be executed when
      expr1 is true but expr2 is false
   }
}
```

Switch Statement

A switch statement allows a variable to be tested for equality against a list of values.

The syntax of the switch statement is −

```
switch (Expression){

   // if expr equals Value1
   case Value1:
      Statement1;
      Statement2;
      break;

   // if expr equals Value2
   case Value2:
      Statement1;
      Statement2;
      break;
      .
      .
   // if expr is other than the specific values above
   default:
      Statement1;
      Statement2;
}
```

Ternary Operator

The ternary operator (?:) is also known as the conditional operator. It can be used as a replacement for an if-else statement.

The syntax of the ternary operator is −

```
(condition) ? true_block: false_block;
```

Loops

C loops are used to execute blocks of one or more statements respectively a specified number of times, or till a certain condition is reached. The following are the loop statements in C language −

- while Loop
- do...while Loop
- for Loop

while Loop

The while loop is an entry-control loop where the condition is checked before executing the loop body.

The syntax of the while loop is −

```
while(test_condition){
   // Statement(s);
}
```

do…while Loop

The do…while loop is an exit control loop where the body of the loop executes before checking the condition.

The syntax of do…while loop is −

```
do{
   // Statement(s);
}while(test_condition);
```

for Loop

The for loop is also an entry-controlled loop where the elements (initialization, test condition, and increment) are placed together to form a for loop inside the parenthesis with the for keyword.

The syntax of the for loop is −

```
for(initialization ; test condition ; increment){
   // Statement (s);
}
```

Jump Statements

Jump statements are used to transfer the flow of the program from one place to another. The following are the jump statements in C language −

- goto Statement
- break Statement
- continue Statement

goto Statement

The goto statement transfers the program's control to a specific label. You need to define a label followed by the colon (:). The goto statement can transfer the program's flow up or down.

The syntax of the goto statement is −

label_name:

```
//Statement(s)
if(test_condition){
   goto label_name;
}
```

break Statement

The break statement can be used with loops and switch statements. The break statement terminates the loop execution and transfers the program's control outside of the loop body.

The syntax of the break statement is −

```
while(condition1){
   . . .
   . . .
   if(condition2)
   break;
      . . .
      . . .
}
```

continue Statement

The continue statement is used to skip the execution of the rest of the statement within the loop in the current iteration and transfer it to the next loop iteration.

The syntax of the continue statement is −

```
while (expr){
   . . .
   . . .
   if (condition)
      continue;
   . . .
}
```

User-defined Functions

The user-defined function is defined by the user to perform specific tasks to achieve the code reusability and modularity.

Example of user-defined function

```
#include <stdio.h>

// Function declaration
int add(int, int);

// Function definition
int add(int a, int b) { return (a + b); }

int main() {
   int num1 = 10, num2 = 10;
   int res_add;

   // Calling the function
   res_add = add(num1, num2);

   // Printing the results
   printf("Addition is : %d\n", res_add);

   return 0;
}
```

Output

```
Addition is : 20
```

Arrays

An array is a collection of data items of similar data type which are stored at a contiguous memory location. The data item may be primary data types (int, float, char), or user-defined types such as struct or pointers can be stored in an array.

C Arrays can be of two types −

- **One-dimensional (1D) Array** − A one-dimensional array is a single list of data items of the same data type.
- **Multi-dimensional Arrays** − A multi-dimensional array such as a two-dimensional array is an array of arrays.

Syntax of Arrays

The following is the syntax of declarations of different types of arrays −

```
type array_name [size1];   // One-dimensional array
type array_name [size1][size2];   // Two-dimensional arrays
type array_name [size1][size2][size3];    // Three-dimensional arrays
```

Example of One-dimensional Array

```
#include <stdio.h>

int main(){
   int numbers[5] = {10, 20, 30, 40, 50};

   int i;  // loop counter

   // Printing array elements
   printf("The array elements are : ");
   for (i = 0; i < 5; i++) {
      printf("%d ", numbers[i]);
   }

   return 0;
}
```

Output

```
The array elements are : 10 20 30 40 50
```

Example of Two-dimensional Arrays

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {
      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Output

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

Strings

C string is a sequence of characters i.e., an array of char data type, terminated by "null character" represented by '\0'. To read and print the string using scanf() and printf() functions, you will have to use the "%s" format specifier.

String Declaration

```
char string_name[size];
```

Reading String

```
scanf("%s", string_name);
```

Printing String

```
printf("%s", string_name);
```

Example of C strings

```
#include <stdio.h>

int main() {
   char name[20];

   printf("Enter a name: ");
   scanf("%s", name);

   printf("You entered: %s", name);

   return 0;
}
```

Strings Functions

C standard library string.h provides various functions to work with the strings. Here is the list of C string functions −

Sr.No.	Function	Description
1	char *strcat	Appends the string pointed to, by *src* to the end of the string pointed to by *dest*.

2	char *strncat	Appends the string pointed to, by *src* to the end of the string pointed to, by *dest* up to n characters long.
3	char *strchr(	Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument *str*.
4	int strcmp	Compares the string pointed to, by *str1* to the string pointed to by *str2*.
5	int strncmp	Compares at most the first n bytes of *str1* and *str2*.
6	int strcoll	Compares string *str1* to *str2*. The result is dependent on the LC_COLLATE setting of the location.
7	char *strcpy	Copies the string pointed to, by *src* to *dest*.
8	char *strncpy	Copies up to n characters from the string pointed to, by *src* to *dest*.
9	size_t strcspn	Calculates the length of the initial segment of str1 which consists entirely of characters not in str2.
10	char *strerror	Searches an internal array for the error number errnum and returns a pointer to an error message string.
11	size_t strlen	Computes the length of the string str up to but not including the terminating null character.
12	char *strpbrk	Finds the first character in the string *str1* that matches any character specified in *str2*.
13	char *strrchr	Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument *str*.
14	size_t strspn	Calculates the length of the initial segment of *str1* which consists entirely of characters in *str2*.
15	char *strstr	Finds the first occurrence of the entire string *needle* (not including the terminating null character) which appears in the string *haystack*.
16	char *strtok	Breaks string *str* into a series of tokens separated by *delim*.
17	size_t strxfrm	Transforms the first **n** characters of the string **src** into current locale and places them in the string **dest**.

Structures

C structures are the collection of different data types. Structures are considered user-defined data types where you can group data items of different data types.

Structure Declaration Syntax

```
struct struct_name {
   type1 item1;
   type2 item2;
   .
   .
}structure_variable;
```

Example of Structure

```
#include <stdio.h>

struct book{
   char title[10];
   char author[20];
   double price;
   int pages;
};

int main(){
   struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};

   printf("Title:  %s \n", book1.title);
   printf("Author: %s \n", book1.author);
   printf("Price:  %lf\n", book1.price);
   printf("Pages:  %d \n", book1.pages);
   printf("Size of book struct: %d", sizeof(struct book));
   return 0;
}
```

Output

```
Title:  Learn C
Author: Dennis Ritchie
Price:  675.500000
Pages:  325
Size of book struct: 48
```

Unions

C union is a user-defined data type that allows to store set of data items of different data types in the same memory location.

Syntax of Union Declaration

```
union [union tag]{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

Example of Union

```
#include <stdio.h>

union Data{
   int i;
   float f;
};

int main(){
   union Data data;

   data.i = 10;
   data.f = 220.5;

   printf("data.i: %d \n", data.i);
   printf("data.f: %f \n", data.f);
   return 0;
}
```

Output

```
data.i: 1130135552
data.f: 220.500000
```

Enumerations (enums)

C enumeration (enum) is an enumerated data type that consists of a group of integral constants.

Syntax of enum Declaration

```
enum myenum {val1, val2, val3, val4};
```

Example of Enumeration (enum)

```
#include <stdio.h>

enum status_codes { OKAY = 1, CANCEL = 0, ALERT = 2 };

int main() {
   // Printing values
   printf("OKAY = %d\n", OKAY);
   printf("CANCEL = %d\n", CANCEL);
   printf("ALERT = %d\n", ALERT);

   return 0;
}
```

Output

```
OKAY = 1
CANCEL = 0
ALERT = 2
```

Pointers

C pointers are derived data types that are used to store the address of another variable and can also be used to access and manipulate the variable's data stored at that location.

Syntax of Pointer Declaration

```
data_type *pointer_name;
```

Syntax of Pointer Initialization

If you are declared a pointer, below is the syntax to initialize a pointer with the address of another variable −

```
pointer_name = &variable_name;
```

Pointer Example

```
#include <stdio.h>

int main() {
   int x = 10;

   // Pointer declaration and initialization
   int * ptr = & x;

   // Printing the current value
   printf("Value of x = %d\n", * ptr);

   // Changing the value
   * ptr = 20;

   // Printing the updated value
   printf("Value of x = %d\n", * ptr);

   return 0;
}
```

Output

```
Value of x = 10
Value of x = 20
```

Type of Pointers

There are various types of pointers in C language. They are −

Dynamic Memory Allocations

Memories for variables are declared at compile-time. C language provides some functions for dynamic memory allocations that allow to allocation of memory for the variables at run time.

The functions for dynamic memory allocation are −

- malloc()
- calloc()
- realloc()
- free()

malloc() Function

The malloc() function allocates the requested memory (number of blocks of the specified size) and returns a pointer to it.

The syntax of malloc() function is −

```
malloc (size_t size);
calloc() Function
```

calloc() Function

The calloc() function allocates the requested memory (number of blocks of the specified size) and returns the void pointer. The calloc() function sets allocated memory to zero.

The syntax of calloc() function is −

```
void *calloc(size_t nitems, size_t size)
```

realloc() Function

The realloc() function attempts to resize the memory block pointed to by a pointer that was previously allocated with a call to malloc() or calloc() function.

The syntax of realloc() function is −

```
void *calloc(size_t nitems, size_t size)
```

free() Function

The free() function deallocates the memory previously allocated by a call to calloc(), malloc(), or realloc().

The syntax of realloc() function is −

```
void *calloc(size_t nitems, size_t size)
```

File Handling

File handling refers to perform various operations on files such as creating, writing, reading, deleting, moving, renaming files, etc. C language provides various functions for file handling.

File Operations

The following are the operations that can perform a file using C language file handling functions −

- Creating a new file
- Opening an existing file
- Writing data to a file
- Appending data to a file
- Reading data from a file
- Renaming a file
- Deleting a file
- Closing a file

File Handling Functions

Following is the list of file handling functions in C −

Function	Description
fopen()	Creates, and opens a file in various modes.
fclose()	Closes a file.
fputc(), fputs(), fprintf()	Writes data to a file.
fgetc(), fgets(), fscanf()	Reads data from a file.
fwrite(), fread()	Write and read data to/from a binary file.
rename()	Renames a file.
remove()	Deleted a file.

Example of File Handling

Here is an example of file handling in C language −

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

int main() {
   FILE *file;
   char file_name[] = "my_file.txt";
   char write_data[100] = "Tutorials Point";
   char read_data[100];

   // Opening file in write mode
   file = fopen("file_name.txt", "w");
   if (file == NULL) {
      printf("Error\n");
      return 1;
   }
   // Writing to the file
   fprintf(file, "%s", write_data);
   // Closing the file
   fclose(file);

   // Again, opening the file in read mode
   file = fopen("file_name.txt", "r");
   if (file == NULL) {
      printf("Error.\n");
      return 1;
   }

   // Reading data from the file
   if (fgets(read_data, 100, file) != NULL) {
      // Printing it on the screen
      printf("File's data:\n%s\n", read_data);
   }
   fclose(file);

   return 0;
}
```

Output

```
File's data:
Tutorials Point
```

Preprocessor Directives

The preprocessor directives are part of pre-compilation and start with the hash (#) character. These directives instruct the compiler to expand include and expand the code before starting the process of compilation.

Here is the list of preprocessor directives −

Directive	Description
# define	Substitutes a preprocessor macro.

#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

Example of Preprocessor Directive

This is an example of #define which is one of the preprocessor directives in C language −

```
#define MAX_ARRAY_LENGTH 20
```

Standard Libraries

Here is the list of commonly used libraries (C header files) −

Header file	Usage
stdio.h	Provides functions for standard input and outputs.
string.h	Provides functions for various string operations.
math.h	Provides function for mathematical operations.
stdlib.h	Provides various utility functions for memory allocations, type conversions, etc.
time.h	Provides date-time related functions.