C Programming Cheatsheet

Essential operations for C programming and development

This cheatsheet provides a quick reference to fundamental C syntax, concepts, and programming techniques, ideal for both beginners and experienced programmers for efficient C development.

Basic Syntax — Core language fundamentals

Data Types — Variables and type system

Control Flow — Loops and conditionals

Functions — Function definitions and calls

Pointers & Arrays — Memory management basics

Basic Syntax & Structure

Hello World Program

Basic structure of a C program.

```
#include <stdio.h>

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

Headers & Preprocessor

Include libraries and use preprocessor directives.

```
#include <stdio.h>    // Standard input/output
#include <stdlib.h>   // Standard library
#include <string.h>   // String functions
#include <math.h>     // Math functions

#define PI 3.14159
#define MAX_SIZE 100
```

Comments

Single-line and multi-line comments.

```
// Single-line comment

/*
Multi-line comment
spans multiple lines
*/

// TODO: Implement feature
/* FIXME: Bug in this section */
```

Main Function

Program entry point with return values.

```
int main() {
    // Program code here
    return 0;  // Success
}

int main(int argc, char *argv[]) {
    // argc: argument count
    // argv: argument values (command line)
    return 0;
}
```

Basic Output

Display text and variables to console.

```
printf("Hello\n");
printf("Value: %d\n", 42);
// Multiple values in one line
printf("Name: %s, Age: %d\n", name, age);
```

Basic Input

Read user input from console.

```
int age;
char name[50];
scanf("%d", &age);
scanf("%s", name);
// Read entire line with spaces
fgets(name, sizeof(name), stdin);
```

Data Types & Variables

Fundamental data types and variable declarations.

01

Primitive Types

Basic data types for storing different kinds of values.

```
// Integer types
int age = 25;
short small_num = 100;
long large_num = 1000000L;
long long huge_num = 9223372036854775807LL;

// Floating-point types
float price = 19.99f;
double precise = 3.14159265359;

// Character and boolean (using int)
char grade = 'A';
int is_valid = 1;  // 1 for true, 0 for false
```

02

Arrays & Strings

Arrays and string handling in C.

```
// Arrays
int numbers[5] = {1, 2, 3, 4, 5};
int matrix[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};

// Strings (character arrays)
char name[50] = "John Doe";
char greeting[] = "Hello";
char buffer[100];  // Uninitialized

// String length and size
int len = strlen(name);
int size = sizeof(buffer);
```

03

Constants & Modifiers

Immutable values and storage modifiers.

```
// Constants
const int MAX_SIZE = 100;
const double PI = 3.14159;

// Preprocessor constants
#define BUFFER_SIZE 512
#define TRUE 1
#define FALSE 0

// Storage modifiers
static int count = 0;     // Static variable
extern int global_var;    // External variable
register int fast_var;    // Register hint
```

Control Flow Structures

Conditional Statements

Make decisions based on conditions.

```
// If-else statement
if (age >= 18) {
    printf("Adult\n");
} else if (age >= 13) {
    printf("Teenager\n");
} else {
    printf("Child\n");
}

// Ternary operator
char* status = (age >= 18) ? "Adult" : "Minor";

// Switch statement
switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break;
    case 'B':
        printf("Good job!\n");
        break;
    default:
        printf("Keep trying!\n");
}
```

For Loops

Iterate with counter-based loops.

```
// Traditional for loop
for (int i = 0; i < 10; i++) {
    printf("%d ", i);
}

// Array iteration
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);
for (int i = 0; i < size; i++) {
    printf("%d ", numbers[i]);
}

// Nested loops
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d,%d ", i, j);
    }
}
```

While Loops

Condition-based iteration.

```
// While loop
int count = 0;
while (count < 5) {
    printf("%d\n", count);
    count++;
}

// Do-while loop (executes at least once)
int input;
do {
    printf("Enter a number (0 to quit): ");
    scanf("%d", &input);
} while (input != 0);
```

Loop Control

Break and continue statements.

```
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        continue; // Skip iteration
    }
    if (i == 7) {
        break;    // Exit loop
    }
    printf("%d ", i);
}

// Nested loops with break
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (i == j) break; // Break inner loop only
        printf("%d,%d ", i, j);
    }
}
```

Functions

Function Declaration & Definition

Create reusable code blocks.

```
// Function declaration (prototype)
int add(int a, int b);
void printMessage(char* msg);

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

void printMessage(char* msg) {
    printf("%s\n", msg);
}

// Function call
int result = add(5, 3);
printMessage("Hello, functions!");
```

Passing Arrays to Functions

Functions that work with arrays.

```
// Array as parameter (pointer)
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Modifying array elements
void doubleValues(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] *= 2;
    }
}
```

Recursive Functions

Functions that call themselves.

```
// Factorial calculation
int factorial(int n) {
    if (n <= 1) {
        return 1;  // Base case
    }
    return n * factorial(n - 1);
}

// Fibonacci sequence
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Function Pointers

Pointers to functions for dynamic behavior.

```
// Function pointer declaration
int (*operation)(int, int);

// Assign function to pointer
operation = add;
int result = operation(5, 3);

// Array of function pointers
int (*operations[])(int, int) = {add, subtract, multiply};
result = operations[0](10, 5);
```

Pointers & Memory Management

Understanding pointers and dynamic memory allocation.

Pointer Basics

Declare and use pointers to access memory addresses.

```
int x = 10;
int *ptr = &x;  // Pointer to x

printf("Value of x: %d\n", x);
printf("Address of x: %p\n", &x);
printf("Value of ptr: %p\n", ptr);
printf("Value pointed by ptr: %d\n", *ptr);

// Modify value through pointer
*ptr = 20;
printf("New value of x: %d\n", x);

// Null pointer
int *null_ptr = NULL;
```

Arrays and Pointers

Relationship between arrays and pointers.

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr;  // Points to first element

// Array notation vs pointer arithmetic
printf("%d\n", arr[2]);   // Array notation
printf("%d\n", *(p + 2)); // Pointer arithmetic
printf("%d\n", p[2]);     // Pointer as array

// Iterate using pointer
for (int i = 0; i < 5; i++) {
    printf("%d ", *(p + i));
}
```

Dynamic Memory Allocation

Allocate and deallocate memory at runtime.

```
#include <stdlib.h>

// Allocate memory for single integer
int *ptr = (int*)malloc(sizeof(int));
if (ptr != NULL) {
    *ptr = 42;
    printf("Value: %d\n", *ptr);
    free(ptr);  // Always free allocated memory
}

// Allocate array dynamically
int *arr = (int*)malloc(10 * sizeof(int));
if (arr != NULL) {
    for (int i = 0; i < 10; i++) {
        arr[i] = i * i;
    }
    free(arr);
}
```

String Pointers

Working with strings and character pointers.

```
// String literals and pointers
char *str1 = "Hello";           // String literal
char str2[] = "World";          // Character array
char *str3 = (char*)malloc(20); // Dynamic string

// String functions
strcpy(str3, "Dynamic");
printf("Length: %lu\n", strlen(str1));
printf("Compare: %d\n", strcmp(str1, str2));
strcat(str2, "!");

// Always free dynamic strings
free(str3);
```

Structures and User-Defined Types

Structure Definition

Define custom data types with multiple fields.

```
// Structure definition
struct Rectangle {
    double width;
    double height;
};

// Structure with typedef
typedef struct {
    char name[50];
    int age;
    double gpa;
} Student;

// Create and initialize structures
struct Rectangle rect1 = {5.0, 3.0};
Student student1 = {"Alice", 20, 3.75};

// Access structure members
printf("Area: %.2f\n", rect1.width * rect1.height);
printf("Student: %s, Age: %d\n", student1.name, student1.age);
```

Nested Structures

Structures containing other structures.

```
typedef struct {
    int day, month, year;
} Date;

typedef struct {
    char name[50];
    Date birthdate;
    double salary;
} Employee;

Employee emp = {
    "John Smith",
    {15, 6, 1985},
    50000.0
};

printf("Born: %d/%d/%d\n",
       emp.birthdate.day,
       emp.birthdate.month,
       emp.birthdate.year);
```

Pointers to Structures

Use pointers to access and modify structures.

```
Student *student_ptr = &student1;

// Access using pointer (two methods)
printf("Name: %s\n", (*student_ptr).name);
printf("Age: %d\n", student_ptr->age);

// Modify through pointer
student_ptr->age = 21;
strcpy(student_ptr->name, "Alice Johnson");

// Dynamic structure allocation
Student *new_student = (Student*)malloc(sizeof(Student));
if (new_student != NULL) {
    strcpy(new_student->name, "Bob");
    new_student->age = 19;
    new_student->gpa = 3.2;
    free(new_student);
}
```

Unions and Enums

Alternative data organization methods.

```
// Union - shared memory space
union Data {
    int integer;
    float floating;
    char character;
};

union Data data;
data.integer = 42;
printf("Integer: %d\n", data.integer);

// Enumeration
enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY, SUNDAY
};

enum Weekday today = FRIDAY;
printf("Today is day %d\n", today);
```

File Input/Output Operations

Handle file reading, writing, and manipulation.

File Reading

Read data from text files.

```
#include <stdio.h>

// Read entire file character by character
FILE *file = fopen("data.txt", "r");
if (file != NULL) {
    int ch;
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }
    fclose(file);
}

// Read line by line
FILE *file2 = fopen("lines.txt", "r");
char buffer[256];
while (fgets(buffer, sizeof(buffer), file2) != NULL) {
    printf("Line: %s", buffer);
}
fclose(file2);

// Read formatted data
FILE *numbers = fopen("numbers.txt", "r");
int num;
while (fscanf(numbers, "%d", &num) == 1) {
    printf("Number: %d\n", num);
}
fclose(numbers);
```

Error Checking

Handle file operations safely.

```
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    printf("Error opening file!\n");
    perror("fopen");  // Print system error message
    return 1;
}

// Check for read errors
if (ferror(file)) {
    printf("Error reading file!\n");
}

// Check for end of file
if (feof(file)) {
    printf("Reached end of file\n");
}

fclose(file);
```

File Writing

Write data to text files.

```
// Write to file
FILE *outfile = fopen("output.txt", "w");
if (outfile != NULL) {
    fprintf(outfile, "Hello, file!\n");
    fprintf(outfile, "Number: %d\n", 42);
    fclose(outfile);
}

// Append to existing file
FILE *appendfile = fopen("log.txt", "a");
if (appendfile != NULL) {
    fprintf(appendfile, "New log entry\n");
    fclose(appendfile);
}

// Write array to file
int numbers[] = {1, 2, 3, 4, 5};
FILE *numfile = fopen("numbers.txt", "w");
for (int i = 0; i < 5; i++) {
    fprintf(numfile, "%d ", numbers[i]);
}
fclose(numfile);
```

Binary File Operations

Read and write binary data efficiently.

```
// Write binary data
Student students[3] = {
    {"Alice", 20, 3.75},
    {"Bob", 21, 3.2},
    {"Charlie", 19, 3.9}
};

FILE *binfile = fopen("students.bin", "wb");
fwrite(students, sizeof(Student), 3, binfile);
fclose(binfile);

// Read binary data
Student loaded_students[3];
FILE *readbin = fopen("students.bin", "rb");
fread(loaded_students, sizeof(Student), 3, readbin);
fclose(readbin);
```

String Manipulation

Essential string handling functions and techniques.

String Functions

Common string operations from string.h library.

```
#include <string.h>

char str1[50] = "Hello";
char str2[] = "World";
char dest[100];

// String length
int len = strlen(str1);
printf("Length: %d\n", len);

// String copy
strcpy(dest, str1);
strncpy(dest, str1, 10); // Copy first 10 chars

// String concatenation
strcat(dest, " ");
strcat(dest, str2);
strncat(dest, "!", 1);   // Append 1 character

// String comparison
int result = strcmp(str1, str2);
if (result == 0) {
    printf("Strings are equal\n");
}
```

String Searching

Find substrings and characters within strings.

```
char text[] = "The quick brown fox";
char *ptr;

// Find first occurrence of character
ptr = strchr(text, 'q');
if (ptr != NULL) {
    printf("Found 'q' at position: %ld\n", ptr - text);
}

// Find last occurrence
ptr = strrchr(text, 'o');
printf("Last 'o' at position: %ld\n", ptr - text);

// Find substring
ptr = strstr(text, "brown");
if (ptr != NULL) {
    printf("Found 'brown' at: %s\n", ptr);
}
```

String Conversion

Convert strings to numbers and vice versa.

```
#include <stdlib.h>

// String to number conversion
char num_str[] = "12345";
char float_str[] = "3.14159";

int num = atoi(num_str);
long long_num = atol(num_str);
double float_num = atof(float_str);

printf("Integer: %d\n", num);
printf("Long: %ld\n", long_num);
printf("Double: %.2f\n", float_num);

// Number to string (using sprintf)
char buffer[50];
sprintf(buffer, "%d", 42);
sprintf(buffer, "%.2f", 3.14159);
printf("String: %s\n", buffer);
```

Custom String Processing

Manual string manipulation techniques.

```
// Count characters in string
int countChar(char *str, char target) {
    int count = 0;
    while (*str) {
        if (*str == target) count++;
        str++;
    }
    return count;
}

// Reverse string in place
void reverseString(char *str) {
    int len = strlen(str);
    for (int i = 0; i < len/2; i++) {
        char temp = str[i];
        str[i] = str[len-1-i];
        str[len-1-i] = temp;
    }
}
```

Compilation & Build Process

Compile and build C programs efficiently.

GCC Compilation

GNU Compiler Collection for C.

```
# Basic compilation
gcc -o program main.c

# With debugging information
gcc -g -o program main.c

# Optimization levels
gcc -O2 -o program main.c

# Multiple source files
gcc -o program main.c utils.c math.c

# Include additional directories
gcc -I/usr/local/include -o program main.c

# Link libraries
gcc -o program main.c -lm -lpthread
```

C Standards

Compile with specific C standard versions.

```
# C90/C89 standard (ANSI C)
gcc -std=c89 -o program main.c

# C99 standard
gcc -std=c99 -o program main.c

# C11 standard (recommended)
gcc -std=c11 -o program main.c

# C18 standard (latest)
gcc -std=c18 -o program main.c

# Enable all warnings
gcc -Wall -Wextra -std=c11 -o program main.c
```

Makefile Basics

Automate compilation with make utility.

```
# Simple Makefile
CC = gcc
CFLAGS = -std=c11 -Wall -g
TARGET = program
SOURCES = main.c utils.c

$(TARGET): $(SOURCES)
	$(CC) $(CFLAGS) -o $(TARGET) $(SOURCES)

clean:
	rm -f $(TARGET)

.PHONY: clean
```

Best Practices & Tips

Write clean, efficient, and maintainable C code.

Naming Conventions

Consistent naming makes code more readable.

```
// Variables and functions: snake_case
int student_count;
double calculate_average(int scores[], int size);

// Constants: UPPER_CASE
#define MAX_BUFFER_SIZE 1024
#define PI 3.14159

// Structures: PascalCase or snake_case
typedef struct {
    char name[50];
    int age;
} Student;

// Global variables: prefix with g_
int g_total_count = 0;

// Function parameters: clear names
void process_data(int *input_array, int array_size);
```

Memory Safety

Prevent common memory-related bugs.

```
// Always initialize variables
int count = 0;        // Good
int count;            // Dangerous - uninitialized

// Check malloc return value
int *ptr = malloc(sizeof(int) * 10);
if (ptr == NULL) {
    printf("Memory allocation failed!\n");
    return -1;
}

// Always free allocated memory
free(ptr);
ptr = NULL;  // Prevent accidental reuse

// Array bounds checking
for (int i = 0; i < array_size; i++) {
    // Safe array access
    array[i] = i;
}
```

Performance Tips

Write efficient C code.

```
// Use appropriate data types
char small_num = 10;   // For small values
int normal_num = 1000; // For typical integers

// Minimize function calls in loops
int len = strlen(str); // Calculate once
for (int i = 0; i < len; i++) {
    // Process string
}

// Use register for frequently accessed variables
register int counter;

// Prefer arrays over dynamic allocation when size is known
int fixed_array[100];  // Stack allocation
// vs
int *dynamic_array = malloc(100 * sizeof(int));
```

Code Organization

Structure code for maintainability.

```
// Header file (utils.h)
#ifndef UTILS_H
#define UTILS_H

// Function prototypes
double calculate_area(double radius);
int fibonacci(int n);

// Structure definitions
typedef struct {
    int x, y;
} Point;

#endif // UTILS_H

// Implementation file (utils.c)
#include "utils.h"
#include <math.h>

double calculate_area(double radius) {
    return M_PI * radius * radius;
}
```

Reference: This cheatsheet covers essential C programming commands and modern practices for efficient software development.