Cheat Sheat : C Syntax

Table of Contents

Variables

- **Declaration**:

```
[type] [name];
```

  - Example:

```
int a;
```

- **Initialization**:

```
[type] [name] = [value];
```

  - Example:

```
int a;
```

- **Multiple Declaration**:

```
[type] [name1], [name2], [name3];
```

  - Example:

```
int a, b, c;
```

- **Multiple Initialization**:

```
[type] [name1] = [value1], [name2] = [value2], [name3] = [value3];
```

  - Example:

```
int a = 1, b = 2, c = 3;
```

- **Constant Declaration**:

```
const [type] [name] = [value];
```

  - Example:

```
const int a = 1;
```

- **Global Variable**: Same as local variable but declared outside of any function (at the top of the file and generally before the main function or in a header file (.h)).

Data Types

- **Basic Data Types**:

  - **Description**: | Type | Description | Size | Range | | --- | --- | --- | --- | | char | Character | 1 byte | -128 to 127 | | unsigned char | Unsigned Character | 1 byte | 0 to 255 | | short int | Short Integer | 2 bytes | -32,768 to 32,767 | | unsigned short int | Unsigned Short Integer | 2 bytes | 0 to 65,535 | | int | Integer | 4 bytes | -2,147,483,648 to 2,147,483,647 | | unsigned int | Unsigned Integer | 4 bytes | 0 to 4,294,967,295 | | long int | Long Integer | 8 bytes | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | | unsigned long int | Unsigned Long Integer | 8 bytes | 0 to 18,446,744,073,709,551,615 | | float | Single Precision Floating Point | 4 bytes | 1.2E-38 to 3.4E+38 | | double | Double Precision Floating Point | 8 bytes | 2.3E-308 to 1.7E+308 | | long double | Extended Precision Floating Point | 10 bytes | 3.4E-4932 to 1.1E+4932 | | void | Represents the absence of type | - | - |
  - **Use**:
    - **char**: The char data type is used to store a single character and also integer values between -128 and 127.

```
char a = 'A';
char b = 65;
```

    - **unsigned char**: The unsigned char data type is used to store a single character, but it can store only positive values between 0 and 255. So, we use unsigned char when we know that the value will be positive like the ASCII value of a character.

```
unsigned char a = 'A'; // 65
```

    - **short int**: The short int data type is used to store integer values between -32,768 and 32,767.

```
short int a = 32767;
```

    - **unsigned short int**: The unsigned short int data type is used to store only positive integer values between 0 and 65,535.

```
unsigned short int a = 65535;
```

    - **int**: The int data type is used to store integer values between -2,147,483,648 and 2,147,483,647.

```
int a = 2147483647;
```

    - **unsigned int**: The unsigned int data type is used to store only positive integer values between 0 and 4,294,967,295.

```
unsigned int a = 4294967295;
```

    - **long int**: The long int data type is used to store integer values between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.

```
long int a = 9223372036854775807;
```

    - **unsigned long int**: The unsigned long int data type is used to store only positive integer values between 0 and 18,446,744,073,709,551,615.

```
unsigned long int a = 18446744073709551615;
```

    - **float**: The float data type is used to store single-precision floating-point values. Give a precision of 6 or 9 decimal places.

```
float a = 3.14;
```

    - **double**: The double data type is used to store double-precision floating-point values. Give a precision of 15 or 17 decimal places.

```
double a = 3.14;
```

- **long double**: The long double data type is used to store extended-precision floating-point values. Give a precision of 19 or 20 decimal places.

```
long double a = 3.14;
```

- **void**: The void data type is generally used to specify the return type of a function that does not return any value. But it can also be used to declare generic pointers.

```
void function() {
    void *ptr;
    printf("Hello World");
}
```

- **Derived Data Types**:
  - **Array**:

```
[type] [name][size];
```

    - Example:

```
int arr[5];
```

  - **Pointer**:

```
[type] *[name];
```

    - Example:

```
int *ptr;
```

  - **Structure**:

```
struct [name] {
    [type1] [member1];
    [type2] [member2];
    ...
};
```

    - Example:

```
struct Point {
    int x;
    int y;
};
```

  - **Union**:

```
union [name] {
    [type1] [member1];
    [type2] [member2];
    ...
};
```

    - Example:

```
union Data {
    int i;
    float f;
    char str[20];
};
```

- **Enumeration**:

```
enum [name] {
    [value1],
    [value2],
    ...
};
```

  - Example:

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

Operators

- **Arithmetic Operators**
  - **+**: Addition
    - Example:

```
int a = 1 + 2; // 3
```

  - **-**: Subtraction
    - Example:

```
int a = 2 - 1; // 1
```

  - ***:** Multiplication
    - Example:

```
int a = 2 * 3; // 6
```

    If both operands are integers, the result will be an integer. If one of the operands is a float, the result will be a float.
  - */*: Division
    - Example:

```
int a = 6 / 2; // 3
```

    If both operands are integers, the result will be an integer. If one of the operands is a float, the result will be a float.
  - **%**: Modulus
    - Example:

```
int a = 6 % 4; // 2
```

- **Relational Operators** The relational operators are used to compare two values. They are used in decision-making and loops.
  - **==**: Equal to
    - Example:

```
int a = 1;
int b = 2;
if (a == b) {
    printf("Equal");
}
```

  - **!=**: Not equal to
    - Example:

```
int a = 1;
int b = 2;
if (a != b) {
    printf("Not Equal");
}
```

- **>**: Greater than
  - Example:

```
int a = 2;
int b = 1;
if (a > b) {
    printf("Greater");
}
```

- **<**: Less than
  - Example:

```
int a = 1;
int b = 2;
if (a < b) {
    printf("Lesser");
}
```

- **>=**: Greater than or equal to
  - Example:

```
int a = 2;
int b = 1;
if (a >= b) {
    printf("Greater or Equal");
}
```

- **<=**: Less than or equal to
  - Example:

```
int a = 1;
int b = 2;
if (a <= b) {
    printf("Lesser or Equal");
}
```

- **Logical Operators** The logical operators are used to combine two or more conditions. They are used in decision-making and loops.
  - **&&**: Logical AND
    - Example:

```
int a = 1;
int b = 2;
if (a == 1 && b == 2) {
    printf("Both are true");
}
```

  - **||**: Logical OR
    - Example:

```
int a = 1;
int b = 2;
if (a == 1 || b == 3) {
    printf("At least one is true");
}
```

  - **!**: Logical NOT
    - Example:

```
int a = 1;
if (!(a == 2)) {
    printf("Not true");
}
```

- **Assignment Operators**
  - **=**: Assign
    - Example:

```
int a = 1;
```

  - **+=**: Add and assign
    - Example:

```
int a = 1;
a += 2; // a = a + 2
```

  - **-=**: Subtract and assign
    - Example:

```
int a = 2;
a -= 1; // a = a - 1
```

  - ***=**: Multiply and assign
    - Example:

```
int a = 2;
a *= 3; // a = a * 3
```

  - **/=**: Divide and assign
    - Example:

```
int a = 6;
a /= 2; // a = a / 2
```

  - **%=**: Modulus and assign
    - Example:

```
int a = 6;
a %= 4; // a = a % 4
```

- **Bitwise Operators** The bitwise operators are used to perform bitwise operations on integers. They are used in low-level programming.
  - **&**: Bitwise AND
    - Example:

```
int a = 5 & 3; // 1
```

  - **|**: Bitwise OR
    - Example:

```
int a = 5 | 3; // 7
```

  - **^**: Bitwise XOR
    - Example:

```
int a = 5 ^ 3; // 6
```

  - **~**: Bitwise NOT
    - Example:

```
int a = ~5; // -6
```

- **<<**: Left shift
  - Example:

```
int a = 5 << 1; // 10
```

- **>>**: Right shift
  - Example:

```
int a = 5 >> 1; // 2
```

Control Statements

- **If Statement**: The if statement is used to execute a block of code only if a condition is true.

```
if (condition) {
    // code
}
```

  - Example:

```
int a = 1;
if (a == 1) {
    printf("True");
}
```

You can also have the single line if statement. But it is recommended to use braces to avoid confusion. You can use it when you have only one statement to execute.

```
if (condition) statement;
```

  - Example:

```
int a = 1;
if (a == 1) printf("True");
```

- **If-Else Statement**: The if-else statement is used to execute a block of code if the condition is true and another block of code if the condition is false.

```
if (condition) {
    // code
} else {
    // code
}
```

  - Example:

```
int a = 1;
if (a == 2) {
    printf("True");
} else {
    printf("False");
}
```

You can also have multiple else-if statements.

```
if (condition1) {
    // code
} else if (condition2) {
    // code
} else {
    // code
}
```

  - Example:

```
int a = 1;
if (a == 2) {
    printf("True");
} else if (a == 1) {
    printf("False");
} else {
    printf("None");
}
```

You can also have the single line if-else statement. But it is recommended to use braces to avoid confusion. You can use it when you have only one statement to execute and that the statement and the condition are short.

```
if (condition) statement1; else statement2;
```

- Example:

```
int a = 1;
if (a == 2) printf("True"); else printf("False");
```

Or you can also use the ternary operator for the same purpose. That is better than the single line if-else statement.

```
condition ? statement1 : statement2;
```

- Example:

```
int a = 1;
a == 2 ? printf("True") : printf("False");
```

- **Switch Statement**: The switch statement is used to execute a block of code based on the value of a variable.

```
switch (variable) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    ...
    default:
        // code
        break;
}
```

- Example:

```
int a = 1;
switch (a) {
    case 1:
        printf("One");
        break;
    case 2:
        printf("Two");
        break;
    default:
        printf("None");
        break;
}
```

The break statement is used to exit the switch statement. If you don't use the break statement, all the cases after the matching case will be executed.

If inside a case you don't have any code to execute, you can use the empty statement.

```
case value:
    ;
    break;
```

You can also have multiple cases for the same code.

```
case value1:
case value2:
    // code
    break;
```

You can also have the switch statement without the variable. In this case, you have to use the goto statement to jump to the case. Goto is generally not recommended to use. Goto is used to jump to a specific line of code like in Assembly language.

```
switch (1) {
    case 1:
        printf("One");
        goto end;
    case 2:
        printf("Two");
        goto end;
    default:
        printf("None");
        goto end;
}
end:
```

If inside a case you have multiple lines of code to execute, you can use the block of code. But it is recommended to use braces to avoid confusion.

```
case value: {
    // code
    // code
    break;
}
```

- **Loops**
  - **While Loop**: The while loop is used to execute a block of code as long as the condition is true.

    ```
    while (condition) {
        // code
    }
    ```

    - Example:

      ```
      int a = 1;
      while (a < 5) {
          printf("%d\n", a);
          a++;
      }
      ```

  - **Do-While Loop**: The do-while loop is used to execute a block of code at least once and then as long as the condition is true.

    ```
    do {
        // code
    } while (condition);
    ```

    - Example:

```
int a = 1;
do {
    printf("%d\n", a);
    a++;
} while (a < 5);
```

- **For Loop**: The for loop is used to execute a block of code a specific number of times.

```
for (initialization; condition; increment/decrement) {
    // code
}
```

  - Example:

```
for (int i = 1; i < 5; i++) {
    printf("%d\n", i);
}
```

  You can also have multiple initializations and multiple increment/decrement statements.

```
for (initialization1, initialization2; condition; increment1, increment2) {
    // code
}
```

  - Example:

```
for (int i = 1, j = 1; i < 5; i++, j++) {
    printf("%d %d\n", i, j);
}
```

  You can also have the empty for loop. But it is recommended to use while loop for this purpose.

```
for (; condition;) {
    // code
}
```

  - Example:

```
int i = 1;
for (; i < 5;) {
    printf("%d\n", i);
    i++;
}
```

  You can also have the infinite for loop. But it is recommended to use while loop for infinite loops.

```
for (;;) {
    // code
}
```

  - Example:

```
for (;;) {
    printf("Infinite\n");
}
```

- **Break Statement**: The break statement is used to exit the loop or switch statement.

```
break;
```

  - Example:

```
for (int i = 1; i < 5; i++) {
    if (i == 3) {
        break;
    }
    printf("%d\n", i);
}
```

- **Continue Statement**: The continue statement is used to skip the current iteration of the loop and continue with the next iteration.

```
continue;
```

  - Example:

```
for (int i = 1; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    printf("%d\n", i);
}
```

- **Goto Statement**: The goto statement is used to jump to a specific line of code.

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

  - Example:

```
int a = 1;
if (a == 1) {
    goto end;
}
printf("Not true");
end:
printf("True");
```

    But in the above example, when a = 1 the code will jump to the end label and execute the code after the end label. So, the output will be "True". But when a = 2 the code will execute the code after the goto statement. So, the output will be "Not trueTrue", because the code after the end label will also be executed. That is why goto is generally not recommended to use.

Functions

- **Declaration**: When you want to declare a function you have to declare it before the main function. The declaration of the function is the prototype of the function. The prototype of the function includes the return type, the name of the function, and the parameters of the function. It is why it is recommended to use the function declaration in a header file (.h).

```
[returnType] [name]([parameters]);
```

  - Example:

```
// Declaration in a header file (.h) like add.h
int add(int a, int b);
```

- **Definition**: The definition of the function is the implementation of the function. It includes the return type, the name of the function, the parameters of the function, and the body of the function. The body of the function is the code that will be executed when the function is called. The definition of the function is generally written after the main function. So you can define the function in any order you want. This is why you have to declare the function before in a header file (.h) and include it in the file where you want to define the function.

```
#include "add.h" // Include the header file where the function is declared
// Definition in the file where you want to define the function like add.c
[returnType] [name]([parameters]) {
    // code
}
```

- Example:

```
int add(int a, int b) {
    return a + b;
}
```

- **Call**: When you want to call a function you have to use the name of the function followed by the parameters of.

```
[name]([parameters]);
```

- Example:

```
#include "add.h" // Include the header file where the function is declared
int main() {
    int a = 1;
    int b = 2;
    int c = add(a, b); // Call the function
    printf("%d\n", c);
    return 0;
}
```

This example works because the function add is declared in the header file add.h and defined in the file add.c. The main function is in the file main.c. The main function includes the header file add.h where the function add is declared. So, the main function knows the prototype of the function add. When the main function is compiled, the compiler knows the prototype of the function add. When the main function is executed, the function add is called. The compiler knows the definition of the function add because it is defined in the file add.c. So, the compiler can execute the function add. When you want to compile a code with multiple files, you have to compile all the files (.c) together. This is why the code above works.

- **Return**: The return statement is used to return a value from the function. The return statement is generally used at the end of the function. When the return statement is executed, the function will be terminated and the control will be returned to the calling function.

```
return [value];
```

- Example:

```
int add(int a, int b) {
    return a + b;
}
```

You can also return nothing from the function. In this case, you have to use the void return type.

```
void [name]([parameters]) {
    // code
    return;
}
```

- Example:

```
void print() {
    printf("Hello World\n");
    return;
}
```

Built-in Functions

- **Input/Output Functions**
  - **printf**: The printf function is used to print the formatted output to the standard output (usually the screen).

```
printf("format", [arguments]);
```

- Example:

```
int a = 1;
printf("Hello World\n");
printf("Value: %d\n", a);
```

- **scanf**: The scanf function is used to read the formatted input from the standard input (usually the keyboard).

```
scanf("format", [arguments]);
```

- Example:

```
int a;
scanf("%d", &a);
```

- **Format**: | Format | Description | Example | | --- | --- | --- | | `%d` or `%i` | Integer | 392 | | `%u` | Unsigned decimal Integer | 7235 | | `%o` | Unsigned octal | 610 | | `%x` | Unsigned hexadecimal integer | 7fa | | `%X` | Unsigned hexadecimal integer (uppercase) | 7FA | | `%f` | Floating point | 392.65 | | `%e` | Scientific notation | 3.9265e+2 | | `%E` | Scientific notation (uppercase) | 3.9265E+2 | | `%g` | Use the shortest representation: `%e` or `%f` | 392.65 | | `%G` | Use the shortest representation: `%E` or `%f` | 392.65 | | `%c` | Character | a | | `%s` | String | sample | | `%p` | Pointer address | b8000000 | | `%n` | Nothing printed | - | | `%%` | Print a % | % |

  They are also other format specifiers like `%5-d` to print an integer with a width of 5 characters. You can also use the flags `+`, `-`, `0`, `#`, and space. You can also use the precision `%.2f` to print a floating-point number with 2 decimal places, etc...

  - **Text Format**: | Format | Description | | --- | --- | | `\n` | New line | | `\t` | Tab | | `\b` | Backspace | | `\r` | Carriage return | | `\\` | Backslash | | `\"` | Double quote | | `\'` | Single quote | | `\0` | Null character |

- **Math Functions** Most of the math functions are defined in the math.h header file. So, you have to include the math.h header file to use the math functions.

- **abs**:
    The abs function is used to return the absolute value of an integer.
    ```c
    abs([value]);
    ```
    - Example:
        ```c
        int a = -1;
        int b = abs(a); // 1
        ```
- **fabs**:
    The fabs function is used to return the absolute value of a floating-point number.
    ```c
    fabs([value]);
    ```
    - Example:
        ```c
        float a = -1.5;
        float b = fabs(a); // 1.5
        ```
- **ceil**:
    The ceil function is used to return the smallest integer value greater than or equal to a floating-point number.
    ```c
    ceil([value]);
    ```
    - Example:
        ```c
        float a = 1.5;
        float b = ceil(a); // 2
        ```
- **floor**:
    The floor function is used to return the largest integer value less than or equal to a floating-point number.
    ```c
    floor([value]);
    ```
    - Example:
        ```c
        float a = 1.5;
        float b = floor(a); // 1
        ```
- **sqrt**:
    The sqrt function is used to return the square root of a floating-point number.
    ```c
    sqrt([value]);
    ```
    - Example:
        ```c
        float a = 4;
        float b = sqrt(a); // 2
        ```
- **pow**:
    The pow function is used to return the value of a number raised to the power of another number.
    ```c
    pow([base], [exponent]);
    ```
    - Example:
        ```c
        float a = 2;
        float b = 3;
        float c = pow(a, b); // 8
        ```
- **log**:
    The log function is used to return the natural logarithm of a floating-point number.
    ```c
    log([value]);
    ```

```
- Example:
    ```c
    float a = 2.71828;
    float b = log(a); // 1
    ```
```

- **String Functions** Most of the string functions are defined in the string.h header file. So, you have to include the string.h header file to use the string functions.
    - **strlen**: The strlen function is used to return the length of a string.

      ```
      strlen([string]);
      ```

        - Example:

          ```
          char str[] = "Hello";
          int len = strlen(str); // 5
          ```

    - **strcpy**: The strcpy function is used to copy a string to another string.

      ```
      strcpy([destination], [source]);
      ```

        - Example:

          ```
          char str1[20];
          char str2[] = "Hello";
          strcpy(str1, str2);
          ```

    - **strcat**: The strcat function is used to concatenate two strings.

      ```
      strcat([destination], [source]);
      ```

        - Example:

          ```
          char str1[20] = "Hello";
          char str2[] = " World";
          strcat(str1, str2);
          ```

    - **strcmp**: The strcmp function is used to compare two strings.

      ```
      strcmp([string1], [string2]);
      ```

        - Example:

          ```
          char str1[] = "Hello";
          char str2[] = "Hello";
          int cmp = strcmp(str1, str2); // 0
          ```

    - **strchr**: The strchr function is used to return the first occurrence of a character in a string.

      ```
      strchr([string], [character]);
      ```

        - Example:

          ```
          char str[] = "Hello";
          char *ptr = strchr(str, 'l'); // llo
          ```

    - **strstr**: The strstr function is used to return the first occurrence of a substring in a string.

      ```
      strstr([string], [substring]);
      ```

        - Example:

```
char str[] = "Hello World";
char *ptr = strstr(str, "World"); // World
```

Pointers

A pointer is a variable that stores the memory address of another variable. Pointers are used to store the address of variables, arrays, functions, etc... Pointers are used to pass the address of variables to functions, to access the memory directly, to allocate memory dynamically, etc...

- **Referencing**: Referencing is the process of obtaining the memory address of a variable. This is done using the address-of operator (&). For example, if you have a variable x, you can obtain its memory address using &x. The result of referencing is a pointer, which holds the memory address of the variable.

```
int x = 10;
int *ptr = &x; // Referencing the variable x

printf("%d\n", x); // 10
printf("%p\n", &x); // 0x7ffebc1f3bfc

printf("%d\n", *ptr); // 10
printf("%p\n", ptr); // 0x7ffebc1f3bfc
```

- **Dereferencing**: Dereferencing is the process of accessing the value stored at a memory address pointed to by a pointer. This is done using the dereference operator (*). For example, if you have a pointer ptr that points to a variable, you can access the value of that variable using *ptr. Dereferencing allows you to manipulate the value stored at a particular memory address indirectly through a pointer.

```
int x = 10;
int *ptr = &x;

printf("%d\n", x); // 10
printf("%p\n", &x); // 0x7ffebc1f3bfc

printf("%d\n", *ptr); // 10
printf("%p\n", ptr); // 0x7ffebc1f3bfc

*ptr = 20; // Dereferencing the pointer and changing the value of x

printf("%d\n", x); // 20
printf("%p\n", &x); // 0x7ffebc1f3bfc

printf("%d\n", *ptr); // 20
printf("%p\n", ptr); // 0x7ffebc1f3bfc
```

- **Pointer Arithmetic**: Pointer arithmetic is the process of performing arithmetic operations on pointers. When you perform arithmetic operations on pointers, the result is determined by the size of the data type the pointer points to. For example, if you have a pointer to an integer, incrementing the pointer by 1 will move it to the next integer in memory, which is 4 bytes away.

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;

printf("%d\n", *ptr); // 10
printf("%d\n", *(ptr + 1)); // 20
printf("%d\n", *(ptr + 2)); // 30
printf("%d\n", *(ptr + 3)); // 40
printf("%d\n", *(ptr + 4)); // 50
```

- **Pointer to Pointer**: A pointer to a pointer is a pointer that holds the memory address of another pointer. Pointer to pointer is used to store the address of a pointer variable.

```
int x = 10;
int *ptr1 = &x;
int **ptr2 = &ptr1;

printf("%d\n", x); // 10
printf("%p\n", &x); // 0x7ffebc1f3bfc

printf("%d\n", *ptr1); // 10
printf("%p\n", ptr1); // 0x7ffebc1f3bfc

printf("%d\n", **ptr2); // 10
printf("%p\n", *ptr2); // 0x7ffebc1f3bfc
printf("%p\n", ptr2); // 0x7ffebc1f3bf8
```

- **Pointer to Function**: A pointer to a function is a pointer that holds the memory address of a function. Pointer to function is used to store the address of a function.

```
int add(int a, int b) {
    return a + b;
}

int (*ptr)(int, int) = add;

printf("%d\n", add(10, 20)); // 30
printf("%d\n", ptr(10, 20)); // 30
```

- **Pointer to Array**: A pointer to an array is a pointer that holds the memory address of the first element of an array. Pointer to array is used to store the address of an array. This is mostly the same thing as pointer arithmetic.
    - **Difference**:
        - **Pointer to array**: Points to the entire array, not just to a single element. The pointer itself knows the size of the array it points to.
        - **Pointer arithmetic**: Involves manipulating the address stored in a pointer based on the size of the data type it points to. It's commonly used for iterating through arrays or navigating memory.

```
int arr[] = {10, 20, 30, 40, 50};
int arr[5] = {1, 2, 3, 4, 5};
int (*ptr)[5] = &arr;

printf("%d\n", (*ptr)[0]); // 10
printf("%d\n", (*ptr)[1]); // 20
printf("%d\n", (*ptr)[2]); // 30
printf("%d\n", (*ptr)[3]); // 40
printf("%d\n", (*ptr)[4]); // 50
```

- **Pointer to Structure**: A pointer to a structure is a pointer that holds the memory address of a structure. Pointer to structure is used to store the address of a structure.

```
struct Point {
    int x;
    int y;
};

struct Point p = {10, 20};
struct Point *ptr = &p;

printf("%d\n", p.x); // 10
printf("%d\n", p.y); // 20

printf("%d\n", ptr->x); // 10
printf("%d\n", ptr->y); // 20
```

  The arrow operator (`->`) is used to access the members of a structure through a pointer to a structure. Because the arrow operator is a shorthand for dereferencing a pointer and accessing a member of a structure. It is equivalent to `(*ptr).x`.

- **Pointer to Union**: A pointer to a union is a pointer that holds the memory address of a union. Pointer to union is used to store the address of a union. Same as the pointer to structure.

```
union Data {
    int i;
    float f;
    char str[20];
};

union Data data;
union Data *ptr = &data;

data.i = 10;
printf("%d\n", data.i); // 10

ptr->i = 20;
printf("%d\n", ptr->i); // 20
```

- **Pointer to Void**: A pointer to void is a pointer that holds the memory address of a variable of any data type. Pointer to void is used to store the address of a variable of any data type.

```
int x = 10;
float y = 20.5;
char z = 'a';

void *ptr1 = &x;
void *ptr2 = &y;
void *ptr3 = &z;

printf("%d\n", *(int *)ptr1); // 10
printf("%f\n", *(float *)ptr2); // 20.5
printf("%c\n", *(char *)ptr3); // a
```

The void pointer is a generic pointer that can point to any data type. But you have to cast the void pointer to the appropriate data type before dereferencing it. The cast operator (`(type)`) is used to convert a void pointer to a specific data type. The syntax is `(type)ptr`.

- **Pointer to Constant**: A pointer to a constant is a pointer that points to a constant value. Pointer to constant is used to store the address of a constant value.

```
int x = 10;
const int *ptr = &x;

printf("%d\n", *ptr); // 10

x = 20;
printf("%d\n", *ptr); // 20

*ptr = 30; // Error
```

The pointer to a constant is used to prevent the value of a constant from being changed through the pointer. The value of the constant can be changed directly, but not through the pointer.

- **Dynamic Pointer**: A dynamic pointer is a pointer that points to memory allocated dynamically. Dynamic pointer is used to store the address of memory allocated dynamically.

```
int *ptr = (int *)malloc(sizeof(int));

*ptr = 10;
printf("%d\n", *ptr); // 10

free(ptr);
```

The malloc function is used to allocate memory dynamically. The sizeof operator is used to determine the size of the data type. The free function is used to deallocate memory.

Arrays

An array is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store multiple values of the same data type.

- **Declaration**: When you want to declare an array you have to specify the data type of the elements and the size of the array.

```
[type] [name][size];
```

  - Example:

```
int arr[5];
```

- **Initialization**: When you want to initialize an array you have to specify the data type of the elements, the size of the array, and the values of the elements.

```
[type] [name][size] = {[values]};
```

  - Example:

```
int arr[5] = {1, 2, 3, 4, 5};
```

  You can also initialize an array without specifying the size.

```
[type] [name][] = {[values]};
```

  - Example:

```
int arr[] = {1, 2, 3, 4, 5};
```

  You can also initialize an array without specifying the values.

```
[type] [name][size];
name[0] = value;
name[1] = value;
```

  - Example:

```
int arr[5];
arr[0] = 1;
arr[1] = 2;
```

- **Access**: When you want to access an element of an array you have to specify the name of the array and the index of the element.

```
[name][index];
```

  - Example:

```
int arr[5] = {1, 2, 3, 4, 5};
printf("%d\n", arr[0]); // 1
printf("%d\n", arr[1]); // 2
```

- **Size**: When you want to get the size of an array you have to use the sizeof operator.

```
sizeof([name]) / sizeof([name][0]);
```

  - Example:

```
int arr[5] = {1, 2, 3, 4, 5};
int size = sizeof(arr) / sizeof(arr[0]); // 5
```

  The sizeof operator is used to determine the size of the element in bytes. So, the size of the array divided by the size of the element gives the number of elements in the array. For example, if the size of the array is 20 bytes and the size of the element is 4 bytes, then the number of elements in the array is 5.

- **Multidimensional Arrays**: A multidimensional array is an array of arrays. It is used to store data in multiple dimensions.
  - **Declaration**: When you want to declare a multidimensional array you have to specify the data type of the elements and the size of each dimension.

```
[type] [name][size1][size2];
```

        - Example:

```
int arr[2][3];
```

    - **Initialization**: When you want to initialize a multidimensional array you have to specify the data type of the elements, the size of each dimension, and the values of the elements.

```
[type] [name][size1][size2] = {[values]};
```

        - Example:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

    - **Access**: When you want to access an element of a multidimensional array you have to specify the name of the array and the indices of the element.

```
[name][index1][index2];
```

        - Example:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
printf("%d\n", arr[0][0]); // 1
printf("%d\n", arr[0][1]); // 2
```

    - **Size**: When you want to get the size of a multidimensional array you have to use the sizeof operator.

```
sizeof([name][dimension]) / sizeof([name][dimension][0]);
```

        - Example:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
int size1 = sizeof(arr) / sizeof(arr[0]); // 2
int size2 = sizeof(arr[0]) / sizeof(arr[0][0]); // 3
```

- **Array of Pointers**: An array of pointers is an array that stores the memory addresses of variables. Array of pointers is used to store the addresses of variables.
    - **Declaration**: When you want to declare an array of pointers you have to specify the data type of the elements and the size of the array.

```
[type] *[name][size];
```

        - Example:

```
int *arr[5];
```

    - **Initialization**: When you want to initialize an array of pointers you have to specify the data type of the elements, the size of the array, and the values of the elements.

```
[type] *[name][size] = {[values]};
```

        - Example:

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
int e = 5;
int *arr[5] = {&a, &b, &c, &d, &e};
```

    - **Access**: When you want to access an element of an array of pointers you have to specify the name of the array and the index of the element.

```
*[name][index];
```

        - Example:

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
int e = 5;
int *arr[5] = {&a, &b, &c, &d, &e};
printf("%d\n", *arr[0]); // 1
printf("%d\n", *arr[1]); // 2
```

- **Size**: When you want to get the size of an array of pointers you have to use the sizeof operator.

```
sizeof([name]) / sizeof([name][0]);
```

  - Example:

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
int e = 5;
int *arr[5] = {&a, &b, &c, &d, &e};
int size = sizeof(arr) / sizeof(arr[0]);
```

Strings

A string is a sequence of characters stored in contiguous memory locations. Strings are used to store text data.

- **Declaration**: When you want to declare a string you have to specify the data type of the elements and the size of the array.

```
[type] [name][size];
```

  - Example:

```
char str[20];
```

- **Initialization**: When you want to initialize a string you have to specify the data type of the elements, the size of the array, and the values of the elements.

```
[type] [name][size] = "[values]";
```

  - Example:

```
char str[20] = "Hello World";
```

You can also initialize a string without specifying the size.

```
[type] [name][] = "[values]";
```

  - Example:

```
char str[] = "Hello World";
```

You can also initialize a string without specifying the values.

```
[type] [name][size];
name[0] = 'H';
name[1] = 'e';
```

  - Example:

```
char str[20];
str[0] = 'H';
str[1] = 'e';
```

As you can see, a string is an array of characters. So, you can access the characters of a string using the same syntax as an array. You can also use the string.h header file to use the string functions.

Structures

A structure is a user-defined data type that allows you to group related data items of different data types. Structures are used to store data in a structured way.

- **Declaration**: When you want to declare a structure you have to specify the keyword struct followed by the name of the structure and the members of the structure. Like a function, you have to declare the structure before the main function. The declaration of the structure is the prototype of the structure. So you have to declare the structure in a header file (.h) and include it in the file where you want to define the structure.

```
// Declaration in a header file (.h) like point.h
struct [name] {
    [type] [member1];
    [type] [member2];
    ...
};
```

  - Example:

```
struct Point {
    int x;
    int y;
};
```

- **Initialization**: When you want to initialize a structure you have to specify the name of the structure, the variable of the structure, and the values of the members of the structure.

```
struct [name] [variable] = {[values]};
```

  - Example:

```
struct Point p = {10, 20};
```

  You can also initialize a structure without specifying the values.

```
struct [name] [variable];
variable.[member1] = [value1];
variable.[member2] = [value2];
```

  - Example:

```
struct Point p;
p.x = 10;
p.y = 20;
```

- **Access**: When you want to access a member of a structure you have to specify the name of the structure followed by the member of the structure.

```
[variable].[member];
```

  - Example:

```
struct Point p = {10, 20};
printf("%d\n", p.x); // 10
printf("%d\n", p.y); // 20
```

  But you can also access the members of a structure using the arrow operator (`->`) when you have a pointer to a structure.

```
[pointer]->[member];
```

- Example:

```
struct Point p = {10, 20};
struct Point *ptr = &p;
printf("%d\n", ptr->x); // 10
printf("%d\n", ptr->y); // 20
```

- **Size**: When you want to get the size of a structure you have to use the sizeof operator.

```
sizeof([name]);
```

- Example:

```
struct Point {
    int x;
    int y;
};
int size = sizeof(struct Point); // 8
```

The size of a structure is the sum of the sizes of its members.

Unions

A union is a user-defined data type that allows you to store different data types in the same memory location. Unions are used to store data in a memory-efficient way. It is similar to a structure, but the memory allocated for a union is the size of the largest member. The use is the same as the structure. Unions are not commonly used because they are not type-safe.

Enumerations

An enumeration is a user-defined data type that allows you to define a set of named integer constants. Enumerations are used to store a set of related constants.

- **Declaration**: When you want to declare an enumeration you have to specify the keyword enum followed by the name of the enumeration and the constants of the enumeration. Like a structure, you have to declare the enumeration before the main function. The declaration of the enumeration is the prototype of the enumeration. So you have to declare the enumeration in a header file (.h) and include it in the file where you want to use it. Because

```
// Declaration in a header file (.h) like color.h
enum [name] {
    [constant1],
    [constant2],
    ...
};
```

- Example:

```
enum Color {
    RED, // 0
    GREEN, // 1
    BLUE // 2
};
```

You can also specify the values of the constants of the enumeration.

```
enum [name] {
    [constant1] = [value1],
    [constant2] = [value2],
    ...
};
```

- Example:

```
enum Color {
    RED = 1,
    GREEN = 4,
    BLUE = 7
};
```

You can also change the default start of the enumeration.

```
enum [name] {
    [constant1] = [value1],
    [constant2],
    ...
};
```

- Example:

```
enum Color {
    RED = 1,
    GREEN, // 2
    BLUE // 3
};
```

And you can also put char constants in the enumeration. NOT STRINGS!

```
enum [name] {
    [constant1] = [value1],
    [constant2] = [value2],
    ...
};
```

- Example:

```
enum Color {
    RED = 'R', // 'R' = 82
    GREEN = 'G', // 'G' = 71
    BLUE = 'B' // 'B' = 66
};
```

- **Initialization**: When you want to initialize an enumeration you have to specify the name of the enumeration, the variable of the enumeration, and the constant of the enumeration.

```
enum [name] [variable] = [constant];
```

- Example:

```
enum Color c = RED; // c = 0
```

The use of enumerations is to make the code more readable and maintainable. Because you can use the constants of the enumeration instead of the integer values.

Typedef

The typedef keyword is used to create an alias for a data type. Typedef is used to define a new data type that can be used to declare variables.

- **Declaration**: When you want to declare a typedef you have to specify the keyword typedef followed by the data type and the alias of the data type.

```
typedef [type] [alias];
```

- Example:

```
typedef int Number;
```

You commonly use typedef with structures and enumerations to make the code more readable and maintainable.

```
typedef struct p {
    int x;
    int y;
} Point;
```

In this example, the alias Point is used to declare variables of the structure p. So you can use the alias `Point` instead of the `struct p`. You can also use typedef with anonymous structures and enumerations.

```
typedef struct {
    int x;
    int y;
} Point;
```

File Handling

File handling is used to read from and write to files. Files are used to store data permanently. Files are used to store data that can be accessed later.

- **Opening a File**: When you want to open a file you have to use the fopen function. The fopen function is used to open a file and return a file pointer.

  ```
  fopen("[filename]", "[mode]");
  ```

  - Example:

    ```
    FILE *file = fopen("file.txt", "w");
    ```

  The mode parameter is used to specify the mode in which the file is opened. There are different modes to open a file: | Mode | Description | | --- | --- | | r | Open a file for reading. The file must exist. | | w | Open a file for writing. If the file exists, it is truncated. If the file does not exist, it is created. | | a | Open a file for appending. If the file exists, the data is written at the end of the file. If the file does not exist, it is created. | | r+ | Open a file for reading and writing. The file must exist. | | w+ | Open a file for reading and writing. If the file exists, it is truncated. If the file does not exist, it is created. | | a+ | Open a file for reading and writing. If the file exists, the data is written at the end of the file. If the file does not exist, it is created. |
- **Closing a File**: When you want to close a file you have to use the fclose function. The fclose function is used to close a file.

  ```
  fclose([file]);
  ```

  - Example:

    ```
    FILE *file = fopen("file.txt", "w");
    fclose(file);
    ```

- **Reading from a File**: When you want to read from a file you have to use the fscanf function. The fscanf function is used to read formatted input from a file.

  ```
  fscanf([file], "format", [arguments]);
  ```

  - Example:

    ```
    FILE *file = fopen("file.txt", "r");
    int a;
    fscanf(file, "%d", &a);
    ```

- **Writing to a File**: When you want to write to a file you have to use the fprintf function. The fprintf function is used to write formatted output to a file.

  ```
  fprintf([file], "format", [arguments]);
  ```

  - Example:

    ```
    FILE *file = fopen("file.txt", "w");
    int a = 10;
    fprintf(file, "%d", a);
    ```

- **Checking the End of a File**: When you want to check the end of a file you have to use the feof function. The feof function is used to check the end-of-file indicator.

```
feof([file]);
```

- Example:

```
FILE *file = fopen("file.txt", "r");
while (!feof(file)) {
    int a;
    fscanf(file, "%d", &a);
}
```

- **Checking the Error of a File**: When you want to check the error of a file you have to use the ferror function. The ferror function is used to check the error indicator.

```
ferror([file]);
```

- Example:

```
FILE *file = fopen("file.txt", "r");
if (ferror(file)) {
    printf("Error\n");
}
```

Preprocessor Directives

Preprocessor directives are used to include header files, define macros, and perform conditional compilation. Preprocessor directives are used to modify the source code before it is compiled.

- **Include**: The include directive is used to include a header file in the source code.

```
#include <[header]>;
```

- Example:

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

You can also include a header file that you have created.

```
#include "[header]";
```

- Example:

```
#include "point.h"
```

- **Define**: The define directive is used to define a macro in the source code.

```
#define [name] [value];
```

- Example:

```
#define PI 3.14159
```

You can also define a macro with arguments.

```
#define [name]([arguments]) [value];
```

- Example:

```
#define SQUARE(x) ((x) * (x))
```

- **If**: The if directive is used to perform conditional compilation in the source code.

```
#if [condition]
...
#endif
```

- Example:

```
#if PI > 3
printf("PI is greater than 3\n");
#endif
```

You can also use the else directive to perform conditional compilation.

```
#if [condition]
...
#else
...
#endif
```

- Example:

```
#if PI > 3
printf("PI is greater than 3\n");
#else
printf("PI is less than or equal to 3\n");
#endif
```

You can also use the elif directive to perform conditional compilation.

```
#if [condition1]
...
#elif [condition2]
...
#endif
```

- Example:

```
#if PI > 3
printf("PI is greater than 3\n");
#elif PI < 3
printf("PI is less than 3\n");
#endif
```

- **Ifdef**: The ifdef directive is used to perform conditional compilation if a macro is defined.

```
#ifdef [name]
...
#endif
```

- Example:

```
#ifdef PI
printf("PI is defined\n");
#endif
```

You can also use the ifndef directive to perform conditional compilation if a macro is not defined.

```
#ifndef [name]
...
#endif
```

- Example:

```
#ifndef PI
printf("PI is not defined\n");
#endif
```

- **Ifndef**: The ifndef directive is used to perform conditional compilation if a macro is not defined.

```
#ifndef [name]
...
#endif
```

  - Example:

```
#ifndef PI
#define PI 3.14159
#endif
```

- **pragma**: The pragma directive is used to provide additional information to the compiler.

```
#pragma [directive];
```

  - Example:

```
#pragma once
```

The once directive is used to include a header file only once in the source code. **IMPORTANT!** The preprocessor directives are executed before the source code is compiled. So, the preprocessor directives are used to modify the source code before it is compiled. So when you whant to define a macro like the struct, enum, function, etc... that you want for your project you have to use the preprocessor directives. This type of macro are defined in the header file (.h) and included in the source file (.c) and they are defined ONCE in the project. So you have two options to define this type of macro:

- **pragma once**: The pragma once directive is used to include a header file only once in the source code.

```
#pragma once
```

The problem with this directive is that it is not supported by all compilers.

- **ifndef**: The ifndef directive is used to perform conditional compilation if a macro is not defined.

```
#ifndef [name]
#define [name]
...
#endif // [name]
```

  - Example:

```
#ifndef POINT_H
#define POINT_H

struct Point {
    int x;
    int y;
};

#endif // POINT_H
```

This is the most common way to define a macro in the header file. The macro is defined only once in the project.

Memory Management

Memory management is used to allocate and deallocate memory dynamically. Memory management is used to allocate memory at runtime and deallocate memory when it is no longer needed.

- **Allocation**: When you want to allocate memory you have to use the `malloc` function. The `malloc` function is used to allocate memory dynamically.

```
malloc([size]);
```

- Example:

```
int *ptr = (int *)malloc(sizeof(int));
```

The size parameter is used to specify the size of the memory to be allocated. The sizeof operator is used to determine the size of the data type.

You can also use the `calloc` function to allocate memory and initialize it to zero.

```
calloc([count], [size]);
```

- Example:

```
int *ptr = (int *)calloc(1, sizeof(int));
```

Because the `calloc` function initializes the memory to zero, it is slower than the `malloc` function. `malloc` function allocates memory but does not initialize it, so you have random values in the memory.

You can also use the `realloc` function to reallocate memory.

```
realloc([ptr], [size]);
```

- Example:

```
int *ptr = (int *)malloc(sizeof(int));
ptr = (int *)realloc(ptr, 2 * sizeof(int));
```

The `realloc` function is used to change the size of the memory block pointed to by the pointer. The size parameter is used to specify the new size of the memory block.

- **Deallocation**: When you want to deallocate memory you have to use the `free` function. The `free` function is used to deallocate memory.

```
free([ptr]);
```

- Example:

```
int *ptr = (int *)malloc(sizeof(int));
free(ptr);
```

The `free` function is used to deallocate memory that was allocated dynamically. The pointer parameter is used to specify the memory block to be deallocated. **IMPORTANT!** When you allocate memory dynamically you have to deallocate it when you no longer need it. If you don't deallocate the memory you will have a memory leak. A memory leak is a situation where memory is allocated but not deallocated, so the memory is not available for other processes. The use of dynamic memory allocation is to allocate memory at runtime when you don't know the size of the memory at compile time. For example, when you want to read a file and store the data in memory, you don't know the size of the data at compile time, so you have to allocate memory dynamically. In a short way, went you need for a short time a memory you have to use the dynamic memory allocation. Went you need for a long time a memory you have to use the static memory allocation (like common variables).

Error Handling

Error handling is used to handle errors that occur during the execution of a program. Error handling is used to prevent the program from crashing when an error occurs.

- **Error Codes**: When you want to handle errors you have to use error codes. Error codes are used to identify errors that occur during the execution of a program.

```
[type] [name] = [value];
```

- Example:

```
int error = 1;
```

You can also use predefined error codes.

```
[type] [name] = [error];
```

- Example:

```
int error = errno;
```

The `errno` variable is used to store the error code of the last error that occurred.

- **Error Handling**: When you want to handle errors you have to use the `perror` function. The `perror` function is used to print an error message to the standard error stream.

```
perror("[message]");
```

  - Example:

```
FILE *file = fopen("file.txt", "r");
if (file == NULL) {
    perror("Error");
}
```

- **Error Messages**: When you want to get an error message you have to use the `strerror` function. The `strerror` function is used to get an error message from an error code.

```
strerror([error]);
```

  - Example:

```
FILE *file = fopen("file.txt", "r");
if (file == NULL) {
    printf("%s\n", strerror(errno));
}
```

- **Exit**: When you want to exit the program you have to use the `exit` function. The `exit` function is used to terminate the program.

```
exit([status]);
```

  - Example:

```
FILE *file = fopen("file.txt", "r");
if (file == NULL) {
    perror("Error");
    exit(1);
}
```

The status parameter is used to specify the exit status of the program. The exit status is used to indicate the success or failure of the program. The exit status 0 indicates success, while the exit status 1 indicates failure.

You can also use the `EXIT_SUCCESS` and `EXIT_FAILURE` macros to specify the exit status.

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

  - Example:

```
FILE *file = fopen("file.txt", "r");
if (file == NULL) {
    perror("Error");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
```

IMPORTANT! When you handle errors you have to print an error message to the standard error stream. The standard error stream is used to print error messages to the console. The standard error stream is different from the standard output stream. The standard output stream is used to print normal output to the console.

Structures of the Project

When you want to create a project you need to have a good structure. In C, the common structure of the project is separated into different directories:

- **include**: The include directory is used to store the header files of the project. The header files contain the declarations of the structures, enumerations, functions, etc...
- **src**: The src directory is used to store the source files of the project. The source files contain the definitions of the structures, enumerations, functions, etc...
- **lib**: The lib directory is used to store the libraries of the project. The libraries contain the compiled code of the project.
- **bin**: The bin directory is used to store the executable files of the project. The executable files contain the compiled code of the project.
- **obj**: The obj directory is used to store the object files of the project. The object files contain the compiled code of the project.
- **doc**: The doc directory is used to store the documentation of the project. The documentation contains the description of the structures, enumerations, functions, etc...
- **test**: The test directory is used to store the test files of the project. The test files contain the test cases of the project.
- **dep**: The dep directory is used to store the dependencies of the project. The dependencies contain the libraries of the project.
- Makefile: The Makefile is used to compile the project. The Makefile contains the rules to compile the project.
- Doxyfile: The Doxyfile is used to generate the documentation of the project. The Doxyfile contains the configuration of the documentation.
- README.md: The README.md is used to describe the project. The README.md contains the description of the project.

This project structure is the complete one. But you can use only the include, src, and Makefile directories to create a simple project. And if you use Visual Studio you don't need to create the project structure because Visual Studio creates it for you. And you don't need to create the Makefile because Visual Studio compiles the project for you.

Makefile

The Makefile is created at the root of the project. The Makefile contains the rules to compile the project.

- **Variables**: The variables are used to store the values that are used in the rules.

```
[name] = [value]
```

  - Example:

```
CC = gcc
CFLAGS = -Wall -Wextra -Werror
```

- **Rules**: The rules are used to compile the project. The rules contain the commands to compile the project.

```
[target]: [dependencies]
    [command]
```

  - Example:

```
main: main.o point.o
    $(CC) $(CFLAGS) -o main main.o point.o
```

  The target is the name of the executable file. The dependencies are the object files. The command is the command to compile the project.
- **Phony**: The phony is used to specify the rules that are not files.

```
.PHONY: [name]
```

  - Example:

```
clean:
    rm -f *.o main

.PHONY: clean
```

  The clean rule is used to clean the project. The clean rule is used to remove the object files and the executable file.
- **Automatic Variables**: The automatic variables are used to store the values that are used in the rules.

```
$@ - The target
$< - The first dependency
$^ - All the dependencies
```

  - Example:

```
main: main.o point.o
    $(CC) $(CFLAGS) -o $@ $^
```

  The automatic variables are used to simplify the rules. The automatic variables are used to avoid repeating the values.
- **Example of a complete Makefile**:

```
CC = gcc
CFLAGS = --std=c99 -Wall -Werror -Iinclude

APP_NAME = main

DEPS = $(wildcard include/*.h)
SRC = $(wildcard src/*.c)
OBJ = $(patsubst src/%.c,obj/%.o,$(SRC))
TESTS = $(wildcard tests/*.c)
TEST_OBJ = $(TESTS:.c=)

obj/%.o: src/%.c $(DEPS)
	mkdir -p obj
	$(CC) -c -o $@ $< $(CFLAGS)

all: compile-app compile-tests

compile-app: $(APP_NAME)

compile-tests: $(TEST_OBJ)

install:
	sudo apt update
	sudo apt upgrade -y
	sudo apt install valgrind -y
	sudo apt install doxygen -y
	python -m venv venv
	venv/bin/pip install pandas
	venv/bin/pip install matplotlib

$(APP_NAME): $(OBJ)
	mkdir -p bin/app
	$(CC) -o bin/app/$(APP_NAME) $^ $(CFLAGS)

tests/%: tests/%.c $(DEPS) $(filter-out src/main.c, $(SRC))
	mkdir -p bin/tests
	$(CC) -o bin/tests/$* $^ $(CFLAGS)

launch-app:
	valgrind --leak-check=full --show-leak-kinds=all -s ./bin/app/$(APP_NAME)

launch-tests:
	for test in bin/tests/*; do $$test upload/map.txt; done

doc:
	doxygen Doxyfile
	cd doc/latex/ && make
	cd ../..
	cp doc/latex/refman.pdf doc/doc.pdf

clean-obj:
	rm -f obj/*.o

clean-exe:
	rm -f bin/app/$(APP_NAME)
	rm -f bin/tests/*

clean-cvs:
	rm -f data/csv/*

clean-plot:
	rm -f data/plot/*

clean-generated:
	rm -f generated/*
```

```
clean-upload:
	rm -f upload/*

clean-doc:
	rm -f doc/*

full-clean: clean-obj clean-exe clean-cvs clean-plot clean-generated clean-upload clean-doc

git-push:
	git add .
	read -p "Enter commit message: " message; \
	git commit -m "$$message"
	git push

.PHONY: all compile-app compile-tests launch-app launch-tests doc clean-obj clean-exe clean-cvs clean-plot clean-generated clean-upl
```

Doxygen

Doxygen is used to generate the documentation of the project. Doxygen is used to generate the documentation of the structures, enumerations, functions, etc...

- **Installation**: Before generating the documentation you have to install Doxygen. You can install Doxygen using the following command:

```
sudo apt update && sudo apt upgrade -y
sudo apt install doxygen -y
```

- **In the Header file**: When you want to generate the documentation you have to use the Doxygen comments. The Doxygen comments are used to describe the structures, enumerations, functions, etc...

```
/**
 * @brief [description]
 * @param [name] [description]
 * @return [description]
 */
```

  - Example:

```
/**
 * @brief The Point structure
 */
struct Point {
    int x; /**< The x-coordinate */
    int y; /**< The y-coordinate */
};

/**
 * @brief The add function
 * @param a The first operand
 * @param b The second operand
 * @return The sum of the operands
 */
int add(int a, int b);
```

  The `@brief` tag is used to specify the description of the structure, enumeration, function, etc... The `@param` tag is used to specify the parameters of the function. The `@return` tag is used to specify the return value of the function.

- **Configuration**: The Doxyfile is used to configure the documentation of the project. The Doxyfile contains the configuration of the documentation.

```
PROJECT_NAME = [name]
OUTPUT_DIRECTORY = [directory]
INPUT = [directory]
```

  - Example:

```
PROJECT_NAME = Project
OUTPUT_DIRECTORY = doc
INPUT = include src
```

The PROJECT_NAME is used to specify the name of the project. The OUTPUT_DIRECTORY is used to specify the directory of the documentation. The INPUT is used to specify the directories of the source files.

- **Generation**: When you want to generate the documentation you have to use the doxygen command. The doxygen command is used to generate the documentation of the project.

```
doxygen [Doxyfile]
```

  - Example:

```
doxygen Doxyfile
```

The Doxyfile is used to configure the documentation of the project. The Doxyfile contains the configuration of the documentation.

Valgrind

Valgrind is used to check the memory leaks of the project.

- **Installation**: Before checking the memory leaks you have to install Valgrind. You can install Valgrind using the following command:

```
sudo apt update && sudo apt upgrade -y
sudo apt install valgrind -y
```

- **Launch**: When you want to check the memory leaks you have to use the valgrind command. The valgrind command is used to check the memory leaks of the project.

```
valgrind --leak-check=full --show-leak-kinds=all -s ./[executable]
```

  - Example:

```
valgrind --leak-check=full --show-leak-kinds=all -s ./bin/main
```

The `--leak-check=full` option is used to check the memory leaks of the project. The `--show-leak-kinds=all` option is used to show all the memory leaks of the project. The `-s` option is used to show the summary of the memory leaks.

GDB

GDB is used to debug the project.

- **Installation**: Before debugging the project you have to install GDB. You can install GDB using the following command:

```
sudo apt update && sudo apt upgrade -y
sudo apt install gdb -y
```

- **Launch**: When you want to debug the project you have to use the gdb command. The gdb command is used to debug the project.

```
gdb ./[executable]
```

  - Example:

```
gdb ./bin/main
```

The `gdb` command is used to launch the GDB debugger. The `./[executable]` is used to specify the executable file of the project.

- **Commands**: When you are in the GDB debugger you have to use the following commands:
  - `run` or `r` - Run the program
  - `break [line]` or `b` - Set a breakpoint at the specified line
  - `print [variable]` - Print the value of the specified variable
  - `next` - Execute the next line of the program
  - `step` or `s` - Execute the next line of the program and step into the function
  - `continue` or `c` - Continue the execution of the program

- `exit` - Quit the GDB debugger