# C Documentation

Group 13

February 18, 2022

## Contents

**1 Lexical Elements** **4**

1.1 Identifiers . . . . . 4

1.2 Keywords . . . . . 4

1.3 Constants . . . . . 4

1.3.1 Integer . . . . . 4

1.3.2 Real Number . . . . . 4

1.3.3 Character . . . . . 4

1.3.4 String . . . . . 5

1.4 Operators . . . . . 5

1.5 Separators . . . . . 5

1.6 White Space . . . . . 5

**2 Data Types** **6**

2.1 Integer . . . . . 6

2.2 Real Number . . . . . 6

2.3 Union . . . . . 7

2.4 Structures . . . . . 8

2.5 Arrays . . . . . 9

2.6 Pointers . . . . . 12

2.7 File . . . . . 12

**3 Expressions and Operators** **14**

3.1 Expression . . . . . 14

3.2 Assignment Operators . . . . . 14

3.3 Incrementing and Decrementing . . . . . . . . . . . . . . . . . . . . . . . 14
3.4 Arithmetic Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.5 Comparison Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.6 Logical Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.7 Bit Shifting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.8 Bitwise Logical Operators . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.9 Pointer Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.10 sizeof Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.11 Type Cast . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.12 Array Subscript . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.13 Function Call as Expressions . . . . . . . . . . . . . . . . . . . . . . . . 17
3.14 The Comma Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.15 Member Access Expressions . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.16 Operator Precedence . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**4 Statements** **21**
4.1 Expression Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2 The `if` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.3 The `switch` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.4 The `while` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.5 The `do` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.6 The `for` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.7 Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.8 The Null Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.9 The `break` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.10 The `continue` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.11 The `return` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.12 The `typedef` Statement . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**5 Functions** **25**
5.1 Declaration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.2 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.3 Calling Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.4 Function Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.5 Calling Functions Through Function Pointers . . . . . . . . . . . . . . . . 27

5.6 main Function . . . . . . . . . . 27
5.7 Recursive Functions . . . . . . . . . . 28
5.8 static Functions . . . . . . . . . . 28

**6 Program Structure and Scope** **29**
6.1 Program Structure . . . . . . . . . . 29
6.2 Scope . . . . . . . . . . 29

**7 Extras** **31**
7.1 Input/Output . . . . . . . . . . 31
7.2 File I/O . . . . . . . . . . 32
7.3 Memory Allocation . . . . . . . . . . 33
7.4 string Library Functions . . . . . . . . . . 35
7.5 math Library Functions . . . . . . . . . . 36

**8 References** **37**

# 1 Lexical Elements

## 1.1 Identifiers

Identifiers are sequences of characters used for naming variables, functions, new data types, and preprocessor macros.
Rules:

- Can include only letters, decimal digits, underscore character.
- The first character of an identifier cannot be a digit.

## 1.2 Keywords

Keywords are part of the programming language and cannot be used for any other purpose. Examples are:
break, case, char, const, continue, default, do, double, else, extern, float, if

## 1.3 Constants

A constant is a literal numeric or character value.

### 1.3.1 Integer

An integer constant is a sequence of digits, with an optional prefix to denote a number base. There are various integer data types, for short integers, long integers, signed integers, and unsigned integers. Examples: 10, 0x15, 230

### 1.3.2 Real Number

A real number constant is a value that represents a fractional number. Examples: 2.11, 3, .7 etc.

### 1.3.3 Character

A character constant is usually a single character enclosed within single quotation marks. The number of character constants in any given character set is limited. The

ASCII character set has 256 characters in it. Examples: ’a’, ’0’, ’A’ etc.

#### 1.3.4 String

A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. All string constants contain a null termination character as their last character.
Examples: "This is a manual", "Hello World!"

## 1.4 Operators

An operator is a special token that performs an operation. Examples: +, +=, * etc.

## 1.5 Separators

Used to separate tokens. Examples : ( ; . : [ ]

## 1.6 White Space

White space is formed by characters such as the space character, the tab character, the newline character. White space is ignored expect when in string and character constants and is used to separate tokens.

# 2 Data Types

## 2.1 Integer

It provides range from 8 bits to 32 bits. Following are the data types with respect to their ranges:
signed char: -128 to 127
unsigned char : 0 to 255
char : same range as signed or unsigned char
short int : -32,768 to 32,768 (16 bit)
unsigned short int : 0 to 65,535 (16 bit)
int : -2,147,483,648 to 2,147,483,647 (32 bit)
unsigned int : 0 to 4,294,967,295 (32 bit)
long int : can be 32 bit or 64 bit long depending on system i.e. similar to int or long long int.
unsigned long int : can be 32 bit or 64 bit long depending on system i.e. similar to unsigned int or unsigned long long int.
long long int : -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (64 bit)
unsigned long long int : 0 to 18,446,744,073,709,551,615 (64 bit)

## 2.2 Real Number

There are three data types that represent fractional numbers. Following is the data type with respect to their sizes:
1. float: The float data type is the smallest of the three floating point types. Its minimum value is stored in the FLT_MIN, and should be no greater than 1e-37. Its maximum value is stored in FLT_MAX, and should be no less than 1e37.
2. double:
The double data type is at least as large as the float type, and it may be larger. Its minimum value is stored in DBL_MIN, and its maximum value is stored in DBL_MAX.
3. long double:
The long double data type is at least as large as the float type, and it may be larger. Its minimum value is stored in LDBL_MIN, and its maximum value is stored in LDBL_MAX.

## 2.3 Union

A union is a custom data type used for storing several variables in the same memory space. We can access any of those variables at any time. Although we can access any of those variables at any time, we should only read from one of them at a time—assigning a value to one of them overwrites the values in the others. Unions are **defined** in the following way:

```
union numbers
{
    double d;
    int i;
};
```

**Declaring unions** can be done in two ways:

1. Along with definition:

```
union numbers
  {
    double d;
    int i;
  } num1,num2;
```

2. After definition

```
union numbers
  {
    double d;
    int i;
  }
 union numbers num1,num2;
```

**Initializing unions:**

```
union numbers
  {
    double d;
    int i;
  }
 union numbers num1 = {d : 3.7};
```

**Accessing unions:**

```
union numbers
  {
    double d;
    int i;
  }
 union numbers num1;
 num1.i = 3;
```

**Size of Unions:**

This size of a union is equal to the size of its largest member.

```
union numbers
  {
    double d;
    int i;
  }
```

The size of the union data type is the same as sizeof (double), because the double type is larger than the int type.

## 2.4 Structures

A structure is a programmer-defined data type made up of variables of other data types.

Declaring, defining, accessing of structs is the same as that of unions as described above. The difference between union and struct is explained by the following example:

```
struct nums
  {
    int x, y;
  };
struct nums num;
num.x = 0;
num.y = 5;


union nums
  {
    int x, y;
  };
struct nums num;
num.x = 0;
num.y = 5;
```

In the first case, x gets value 0 and y gets value 5 whereas in the second case, y overrides the value of the x member. This is because, in union, the members share the same memory location while in struct every member is assigned a different memory location.

**Size of Structures**:
The size of a structure type is equal to the sum of the size of all of its members, possibly including padding to cause the structure type to align to a particular byte boundary.

## 2.5 Arrays

An array is a data structure that lets you store one or more elements consecutively in memory. In C, array elements are indexed beginning at position zero, not one.

**Declaring arrays**:
We declare an array by specifying the data type for its elements, its name, and the number of elements it can store. Example:

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

**Declaring arrays**:
We can initialize the elements in an array when you declare it by listing the initializing values, separated by commas, in a set of braces. Here is an example:

```
int arr[3] = { 0, 1, 2 };
```

**Accessing arrays**:
access the elements of an array by specifying the array name, followed by the element index, enclosed in brackets. Example:

```
int arr[0] = 0;
```

**Multidimensional arrays**:
Declaring multidimensional array can be done by adding an extra set of brackets and array lengths for every additional dimension. Example:

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

**Arrays as strings**:
Arrays can be used to strore chars and thus forming a string. Example:

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

**Arrays of unions**:
We can create an array of a union type just as you can an array of a primitive data type.

```
union nums
{
    int i;
    long long l;
 };
union nums arr_union[2];
```

Initialization:
We can initialize the first members of the elements of a nums array:

```
union nums arr_union[2] = { {3}, {4} };
```

**Arrays of structs**:
We can create an array of a structure type just as you can an array of a primitive data type.

```
struct nums
{
    int i;
    long long l;
 };
struct nums arr_struct[2];
```

Initialization:
We can initialize the members of the elements of a nums array:

```
struct nums arr_struct[2] = { {1,2}, {3,4}};
```

Accessing:

```
struct nums arr_struct[2];
 arr_struct[0].i = 2;
 arr_struct[0].l = 3;
```

## 2.6 Pointers

Pointers hold memory addresses of stored constants or variables.

**Declaring Pointers**:
We declare a pointer by specifying a name for it and a data type. Example: int *ptr;
**Initializing Pointers**:
We can initialize a pointer when we first declare it by specifying a variable address to store in it. Example:

```
int i;
int *ptr = &i;
```

**Pointers to Unions**:
We can create a pointer to a union type as follows:

```
union numbers
  {
    int i;
    float f;
  };
union numbers foo = {4};
union numbers *number_ptr = &foo;
```

**Pointers to Structures**:
We can create a pointer to a structure type as follows:

```
struct nums
  {
    int x,y;
  };
struct nums num = {4, 5};
struct nums *numbers = &num;
```

## 2.7 File

A file is a type of structure typedef as FILE. The file object contains all the information necessary to control the file stream. The file represents a sequence of bytes which can be either a text file or a binary file. For example:

```
FILE f1,f2;
```

# 3 Expressions and Operators

## 3.1 Expression

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values.

## 3.2 Assignment Operators

Assignment operators are used to store values in variables. The = operator stores the value of a right operand in the variable specified by left operand. For example,

```
int x = 1;
float y = 5.2;
```

## 3.3 Incrementing and Decrementing

The increment operator ++ adds 1 to its operand. The increment operator can be applied either before or after the operand. For example,

```
int x = 2;
int y = 7;
x++; // now x is 3
++y; // now y is 8
```

Likewise, the decrement operator – subtracts 1 from its operand.

## 3.4 Arithmetic Operators

Operators can be used for standard arithmetic operations: addition, subtraction, multiplication, division, modular division and negation. Addition and subtraction can also be done with memory pointers. Integer division of positive values truncates towards 0. The modulus operator % is used to obtain the remainder by performing integer division of its 2 operands.

## 3.5 Comparison Operators

Comparison operators are used to determine the relation between two operands, i.e., whether they are equal to each other, one is larger than the other one is smaller than the other. It gives result as 0(false) or 1(true).
The equal equal to operator == checks equality of two operands.
The not equal to operator != checks for inequality of two operands.
The greater than operator > checks if one operand is greater than the other operand.
The greater than or equal to operator >= checks if one operand is greater than or equal to the other operand.
The less than operator < checks if one operand is less than the other operand.
The less than or equal to operator <= checks if one operand is less than or equal to the other operand.

## 3.6 Logical Operators

Logical operators are used to find the truth value of a pair of operands. A nonzero expression is considered to be true while an expression that evaluates to zero is considered false.
The logical conjunction operator && checks if two expressions are both true. If the first expression is false, then the second expression is not evaluated.
The logical disjunction operator || checks if at least one of two expressions it true. If the first expression is true, then the second expression is not evaluated.
The logical negaion operator ! is used to flip the truth value of the expression.

## 3.7 Bit Shifting

The left-shift operator « is used to shift its first operand's bits to the left. The second operator denotes the number of bit places to shift.

```
x = 1;
x << 3;
// Now x becomes 8
```

Similarly the right-shift operator is used to shift its first operand's bits to the right.

## 3.8 Bitwise Logical Operators

Bitwise logical operators are used for performing bitwise conjunction, inclusive disjunction, exclusive disjunction, and negation.
Biwise conjunction operator & checks each bit in its two operands, and when two corresponding bits are both 1, the resulting bit is 1, otherwise resulting bit is 0.
Bitwise inclusive disjunction operator | checks each bit in its two operands, and when two corresponding bits are both 0, the resulting bit is 0, otherwise resulting bit is 1.
Bitwise exclusive disjunction operator ĉhecks each bit in its two operands, and when two corresponding bits are different, the resulting bit is 1, otherwise resulting bit is 0.
Bitwise negation operator reverses each bit in its operand

## 3.9 Pointer Operators

The address operator & is used to obtain the memory address of an object. To obtain the address of a function, it is not mandatory to use this operator. We can also obtain the address of a label with the label address operator &&. Given a memory address stored in a pointer, you can use the indirection operator * to obtain the value stored at the address (**dereferencing**). Some examples of the usage of pointer operator are as follows:

```
int x = 1;
int *ptr = &x; // pointer to x

extern int foo (void);
int (*fp) (void) = foo; // fp points to foo
```

## 3.10 sizeof Operator

The sizeof operator is used to obtain the size(in bytes) of the data type of its operand. The operand may be a type specifier (such as int or float), or any valid expression.

When the operand is a type name, it must be enclosed in parentheses. The result of the sizeof operator is of a type called size_t. For example,

```
size_t a = sizeof(int);
size_t b = sizeof(8.2);
size_t c = sizeof(a);
```

## 3.11 Type Cast

A type cast consists of a type specifier(for example- int, float) enclosed in parentheses, followed by an expression which is used to explicitly make an expression to be of a specified data type. For proper casting, expression should also be enclosed that follows the type specifier in parentheses.

## 3.12 Array Subscript

Elements of an array can be accessed by specifying the name of the array and the index of the element (zero-indexing followed) enclosed in square brackets. For example:

```
A[0] = 5; //The first element of array A is assigned a value of 5.
```

The array subscript expression `A[i]` is defined as being identical to the expression `(*((A)+(i)))`.

## 3.13 Function Call as Expressions

A call to any function which returns a value is an expression. For example:

```
int function(void);
int x = function() + 2;
```

## 3.14 The Comma Operator

The comma operator is used to separate 2 expressions. Commonly the comma operator is used in for statements to set, monitor and modify multiple control statements. For example,

```
for(x = 1, y = 5; x <= 5 && y >= 1; x++, y--){
    ...
}
```

## 3.15 Member Access Expressions

Member access operator . is used to access the members of a structure or union variable. The name of the structure should be written on the left side of the operator and the name of the member should be written on the right side of the operator. For example:

```
struct S
{
    int x;
    float y;
};
struct S my_struct;
my_struct.x = 1;
my_struct.y = 2.5;
```

Indirect member access operator -> can be used to access the members of a structure or union variable via a pointer. `x->y` is equivalent to `(*x).y`
For example:

```
struct S
{
    int a;
    float b;
};
```

```
struct S st;
struct S *st_ptr = &st;
st_ptr->a = 1;
st_ptr->b = 2.5;
```

## 3.16 Operator Precedence

When an expression contains multiple operators, the operators are grouped based on rules of precedence. A list of types of expressions, in order of highest precedence first, is presented as follows:

1. Function calls, array subscripting, and membership access operator expressions
2. Unary operators, including logical negation, bitwise complement, increment, decrement, unary positive, unary negative, indirection operator, address operator, type casting, and sizeof expressions
3. Multiplication, division, and modular division expressions
4. Addition and subtraction expressions
5. Bitwise shifting expressions
6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions
7. Equal-to and not-equal-to expressions
8. Bitwise AND expressions
9. Bitwise exclusive OR expressions
10. Bitwise inclusive OR expressions
11. Logical AND expressions
12. Logical OR expressions
13. Conditional expressions (evaluated right to left)

14. All assignment expressions, including compound assignment (evaluated right to left)
15. Comma operator expressions

# 4 Statements

## 4.1 Expression Statements

They are expressions which have a semicolon at the end. For example,

```
4 + 3;
y = x + 5;
```

Expression statements can be used to store values, call functions etc.

## 4.2 The `if` Statement

The `if` statement is used to conditionally execute a part of the program. The general form of a `if-else` block is as follows:

```
if (test)
    then-statement
else
    else-statement
```

The `then-statement` is executed if the `test` condition evaluates to true, otherwise the `else-statement` is executed. The `else` clause is optional and can be omitted. Furthermore, a series of `if, else if` clauses can be used to test for multiple conditions.

## 4.3 The `switch` Statement

The `switch` statement is used to compare expressions and execute a series of sub-statements based on the comparisons. The general form of a `switch` statement is as follows:

```
switch (test)
{
    case compare-1
        if-equal-statement-1
    case compare-2:
        if-equal-statement-2
    ...
    default:
        default-statement
}
```

The `switch` statement compares test to each of the compare expressions, until it finds one that is equal to test. Then, the statements following the successful case are executed. The default case is optional and can be omitted.

## 4.4 The `while` Statement

The `while` statement is a loop statement which contains an exit condition at the beginning of the loop. The general form of the `while` statement is as follows:

```
while (test)
    statement
```

The `test` condition is evaluated first. If `test` evaluates to true, `statement` is executed, and then `test` is evaluated again. `statement` continues to execute repeatedly as long as test is true after each execution of `statement`.

## 4.5 The `do` Statement

The `do` statement is a loop statement which contains an exit condition at the end of the loop. The general form of the do statement is as follows:

```
do
    statement
```

```
                while (test);
```

The `statement` is executed first. After that, `test` is evaluated. If `test` is true, then `statement` is executed again. `statement` continues to execute repeatedly as long as `test` is true after each execution of statement.

## 4.6 The `for` Statement

The `for` statement is a loop statement which contain 3 expressions - variable initialization, condition testing and variable modification. For statements are used to make counter-controlled loops. The general form of the for statement is as follows:

```
for (initialize; test; step)
    statement
```

First, the `initialize` expression is evaluated. Then it evaluates the expression `test`. If `test` is false, then the loop ends and program control resumes after statement. Otherwise, if `test` is true, then `statement` is executed. Finally, `step` is evaluated, and the next iteration of the loop begins with evaluating `test` again.

## 4.7 Blocks

A block is a set of zero or more statements enclosed in braces. Block is often used as the body of an if statement or a loop statement, to group statements together. Variables declared inside a block are local to the block.

## 4.8 The Null Statement

The `null` statement is only a semicolon. It does not do anything. It is often used in the body of loops, especially as one of the conditions in a `for` loop.

## 4.9 The `break` Statement

The `break` statement is used to terminate a `while, do, for`, or `switch` statement. When a `break` statement is used inside a loop or `switch` statement which itself is inside another loop or `switch` statement, the `break` only terminates the innermost loop or `switch` statement.

## 4.10 The `continue` Statement

The `continue` statement is used in loops to terminate an iteration of the loop and begin the next iteration. When a `continue` statement is used inside a loop which itself is inside another loop, then it affects only the innermost loop.

## 4.11 The `return` Statement

The `return` statement is used to end the execution of a function and return control to the function that called it. The return value is an optional statement to return. In void functions, it is invalid to return an expression, however the `return` statement can be used without a return value.

## 4.12 The `typedef` Statement

The `typedef` statement is used to create aliases for data types. The general format of a `typedef` statement is as follows:

```
typedef old-type-name new-type-name
```

# 5 Functions

Functions are used to separate the program into different sub-parts, so that each part reflects a particular task.

## 5.1 Declaration

A function declaration is used to explicitly refer to a function present. It is not compulsory to write a function declaration, but not mentioning it might lead for the function not be recognized.
Function declaration is a simple statement of the form:

```
return-type  function-name  (parameters-list);
```

`return-type` refers to the data type of value returned by the function. 'void' to be used if function does not return anything.
`function-name` is the name of the function which is an identifier.
In brackets, we can have zero or more parameters. In function declaration, we only require to mention the data-type of the parameters. But we can also mention the parameter names.
Example,

```
int multiply(int, int);
```

## 5.2 Definitions

Function definitions defines what a function actually does. It consists of return-type, function name, the parameters list, and a body which consists of sequence of different statements. The body is enclosed between curly braces, and this in simple terms is called **blocks**.
Syntax below:

```
return-type  function-name (parameters-list){
    function-body
}
```

`return-type`, `function-name` and `parameters-list` is same as the one in declaration, except that in parameters-list parameter's name must be present.
Example,

```
int multiply (int x, int y){
    return x * y;
}
```

## 5.3 Calling Functions

The function can be called by simply mentioning the function-name and by passing the appropriate parameters.
A function call can be complete statement or can a sub-expression. For example,

```
multiply (3,8);
```

is a complete statement, whereas in:

```
int result = multiply(3,8);
```

function call is in the form of sub-expression.

## 5.4 Function Parameters

Function parameters are passed to the function when the function is called. A function parameter can a literal value, a value stored in a variable, an address in memory or a complex expression combining all these.
The value passed is stored as a local copy in the function. Hence we can not change the value which is passed by changing the local copy in the function.
For changing a value passed, we pass the values using pointers. In this way, we can change the values passed directly where they are stored.
A simple example of this is when we pass an array to the function.

```
int no_of_elements (int a[]);
```

Here the value passed is a pointer to the first element of the array.

## 5.5 Calling Functions Through Function Pointers

You can call a function identified by a pointer. The operator * is optional when doing this.

```
#include<stdio.h>
void foo(int i){
    printf("foo %d!\n", i);
}
void bar(int i){
    printf("%d bar!\n", i);
}
void message(void (*func)(int), int times){
    for(int j = 0; j < times; ++j)
        func(j);
}
void example(int want_foo){
    void (*pf)(int) = bar;
    if(want_foo) pf = foo;
    message(pf,5);
}
```

## 5.6 main Function

Every program has atleast one function which is `main` function. This is where the program starts executing.
We do not need any declaration for this program but definition of this program is required.
The value returned by the `main` function is always `int`, which indicates the program's exit status.
The `return-type` of the `main` function can be void or int, but not any other type.
Returning the value of 0 means success. If we don't return or we return without giving any value, then it means 0 is returned.
Parameters are generally not required for the main function, but for accepting command line arguments we need two parameters, namely "`int argc`" and "`char *argv[]`".

`argc` represents the number of arguments.
`argv` is an array of consisting all those arguments in string format.

## 5.7 Recursive Functions

We can also write recursive functions, where the function calls itself. For example, the factorial function:

```
int factorial(int n){
    if(n < 1) return 1;
    else
        return (n * factorial(n-1));
}
```

Recursive function can be indefinitely recursive. Hence we need to take care of that.

## 5.8 static Functions

A function can be defined as static to make the scope of calling the function limited to the source file the function is present in.

```
static int multiply(int x, int y){
    return x*y;
}
```

This is useful if we don't want a function to be used outside the source file it is defined in.

# 6 Program Structure and Scope

## 6.1 Program Structure

A C program may exist entirely within a single source file, but more commonly, any non-trivial program will consist of several custom header files and source files, and will also include and link with files from existing libraries.
By convention, header files (with a ".h" extension) contain variable and function declarations, and source files (with a ".c" extension) contain the corresponding definitions. Source files may also store declarations, if these declarations are not for objects which need to be seen by other files. However, header files almost certainly should not contain any definitions.

If we require a function which is defined in a source file to be used in other source files, then we require header files, to communicate that the definition of function exists and where it exists. For eg.

```
/* multiply.h */
int multiply(int x, int y)

/* multiply.c */
#include "multiply.h"
...
int multiply(int x, int y){
    return x*y;
}
...
```

Now this function `multiply` can be used in other source files by simple including the `multiply.h` header file.

## 6.2 Scope

Scope refers to the parts of the program where a declared object can be "seen". A declared object can be visible only within a particular function, or within a particular file, or may be visible to an entire set of files by way of including header files and using `extern` declarations.

Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions, but are not visible outside of the file.

Declarations made within functions are visible only within those functions.
A common example is that a declaration is not visible to the declarations that came before it.

# 7 Extras

## 7.1 Input/Output

```
int printf ( const char * format, ... );
```

The `printf` function is used to write the C string pointed by format to the standard output (`stdout`). If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

| **Specifier** | **Output** |
|---|---|
| d | Signed decimal integer |
| u | Unsigned decimal integer |
| o | Unsigned octal |
| x | Unsigned hexadecimal integer |
| f | Decimal floating point |
| c | Character |
| s | String of characters |
| p | Pointer address |
| % | A single % is printed |

The format specifier can be of the form

```
% [width] [.precision] specifier
```

where, **width** is the minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger,
and **precision** specifies the minimum number of digits to be written for integer specifiers (`d, o, u, x`). If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. For `f` format specifier, it is the number of digits to be printed after the decimal point. For `s` format specifier, this is the maximum number of characters to be printed.
On success, the total number of characters written is returned.

```
int scanf ( const char * format, ... );
```

Similarly, the `scanf` function reads data from `stdin` and stores them according to the parameter format into the locations pointed by the additional arguments.
The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the format string.

## 7.2 File I/O

The `fopen()` function is used to create a new file or to open an existing file. The prototype of this function call is as follows:

```
FILE *fopen( const char * filename, const char * mode );
```

`mode` can take any of the following values:

1. `r` : Opens an existing text file for reading purpose.
2. `w` : Opens a text file for writing. If it does not exist, then a new file is created. Here the program will start writing content from the beginning of the file.
3. `a` : Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here the program will start appending content in the existing file content.
4. `r+` : Opens a text file for both reading and writing.
5. `w+` : Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
6. `a+` : Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

To close a file, the `fclose()` function is used. The prototype of this function is

```
int fclose( FILE *fp );
```

The `fclose( )` function returns zero on success, or `EOF` if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The `EOF` is a constant defined in the header file `stdio.h`.

```
int fprintf(FILE *fp,const char *format, ...)
```

The function `fprintf()` writes the string `s` to the output stream referenced by `fp`. It returns a non-negative value on success, otherwise `EOF` is returned in case of any error.

```
int fscanf(FILE *fp, const char *format, char *buf)
```

The functions `fscanf()` reads from the input stream referenced by `fp` till it encounters the first space character. It copies the read string into the buffer `buf`, appending a null character to terminate the string.

## 7.3 Memory Allocation

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

| **Specifier** | **Output** |
|---|---|
| malloc() | allocates single block of requested memory. |
| calloc() | allocates multiple block of requested memory. |
| realloc() | reallocates the memory occupied by malloc() or calloc() functions. |
| free() | frees the dynamically allocated memory. |

**malloc():**
It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.The syntax of malloc() function is given below:

```
ptr=(cast-type*)malloc(byte-size)
```

Example:

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

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

**calloc():**
It initially initializes all bytes to zero. It returns NULL if memory is not sufficient. The syntax of calloc() function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

**realloc():**
If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size. Syntax is:

```
ptr=realloc(ptr, new-size)
```

**free():**
The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit. Syntax is:

```
free(ptr)
```

## 7.4 string Library Functions

The functions present in the string.h header are as follows:

1. void ***memset**(void *str, int c, size_t n): Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument str.
2. char ***strcat**(char *dest, const char *src): Appends the string pointed to, by src to the end of the string pointed to by dest.
3. char ***strncat**(char *dest, const char *src, size_t n): Appends the string pointed to, by src to the end of the string pointed to, by dest up to n characters long.
4. char ***strchr**(const char *str, int c): Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument str.
5. int **strcmp**(const char *str1, const char *str2): Compares the string pointed to, by str1 to the string pointed to by str2.
6. int **strncmp**(const char *str1, const char *str2, size_t n): Compares at most the first n bytes of str1 and str2.
7. char ***strcpy**(char *dest, const char *src): Copies the string pointed to, by src to dest.
8. char ***strncpy**(char *dest, const char *src, size_t n): Copies up to n characters from the string pointed to, by src to dest..
9. char ***strerror**(int errnum): Searches an internal array for the error number errnum and returns a pointer to an error message string.
10. size_t **strlen**(const char *str): Computes the length of the string str up to but not including the terminating null character.
11. char ***strstr**(const char *haystack, const char *needle): Finds the first occurrence of the entire string needle (not including the terminating null character) which appears in the string haystack.

## 7.5 math Library Functions

All the functions in the math Library takes **double** as input and gives **double** as the result.
Functions are given below:

1. double **acos**(double x): Returns the arc cosine of x in radians.
2. double **asin**(double x): Returns the arc sine of x in radians.
3. double **atan**(double x): Returns the arc tangent of x in radians.
4. double **cos**(double x): Returns the cosine of x in radians.
5. double **cosh**(double x): Returns the hyperbolic cosine of x in radians.
6. double **sin**(double x): Returns the sine of x in radians.
7. double **sinh**(double x): Returns the hyperbolic sine of x in radians.
8. double **tanh**(double x): Returns the hyperbolic tangent of x in radians.
9. double **exp**(double x): Returns the value of **e** raised to the xth power.
10. double **log**(double x): Returns the natural logarithm (base-e) of x.
11. double **log10**(double x): Returns the common logarithm (base-10) of x.
12. double **pow**(double x, double y): Returns x raised to the power of y.
13. double **sqrt**(double x): Returns the square root of x.
14. double **ceil**(double x): Returns the smallest integer value greater than or equal to x.
15. double **fabs**(double x): Returns the absolute value of x.
16. double **floor**(double x): Returns the largest integer value less than or equal to x.
17. double **fmod**(double x, double y): Returns the remainder of x divided by y.

## 8 References

- https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html
- https://www.tutorialspoint.com/c_standard_library/string_h.htm
- https://www.javatpoint.com/dynamic-memory-allocation-in-c
- https://www.tutorialspoint.com/c_standard_library/math_h.htm
- https://www.geeksforgeeks.org/data-type-file-c/
- https://www.cplusplus.com/
- https://www.tutorialspoint.com/cprogramming/c_file_io.htm