C Programming Basics

Compiler version

- MSVC
- GCC

Language standard

- C++17 *github
- C11
- For Visual Studio 2019, the version information is given in

> Project\Properties\Configuration Properties\C/C++\Language
> *C++ Language Standard

References

- *programiz: introduction
- *tutorialspoint: introduction
- *geeksforgeeks: details
- *cprogramming: details

Visual Studio

Troubleshooting

- C1041 error: cannot open program database

> Add `C:\Program Files (x86)\Microsoft Visual Studio\` to the anti-virus exception.

Custom Notations

- Though these can be used inside code boxes, these are *irrelevant* to C syntax.

notation	meaning
`<==>`	*equivalence*
`<=>`	the two items have the *same value*.
`<!=>`	the two items are different.

Program Structure

- A C program basically consists of the following parts:
  - Preprocessor Commands

- Functions
- Variables
- Statements & Expressions
- Comments

Tokens

- A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:
  - Keywords
  - Identifiers
  - Constants
  - Strings
  - Special Symbols
  - Operators

Keywords

- *programiz
- Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier.
- List of keywords *programiz

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

Identifiers

- *programiz
- Identifier refers to name given to entities such as variables, functions, structures etc. Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For instance, in `int n;`, `n` is an identifier.
- Identifier names must be different from keywords. `int` cannot be used as an identifier because it is a keyword.

- Rules:
  - The first letter of an identifier should be either a letter or an underscore.
  - The identifier must not exceed *31 characters*, or it may cause problems in some compilers.

Constants

- Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called *literals*. *programiz
- The *values* assigned to each constant variables are referred to as the *literals*. Generally, both terms, constants and literals are used *interchangeably*. For example, `const int = 5;` is a constant expression and the value 5 is referred to as constant integer literal. *geeksforgeeks
- Two ways to define constants: *geeksforgeeks
  - By using the keyword `const`

    ```
    const double PI = 3.14159;
    ```

  - By using `#define` *preprocessor* directive

    ```
    #define PI 3.14159
    ```

  The two methods of definition bear different properties. The former one has a *data type*, whereas the latter one is a *preprocessor macro* that is not allotted a memory. Furthermore, the latter one triggers *conditional directives* such as `#ifdef` and `#if defined(PI)` in contrast to the former one.

- It is recommended to capitalize the name of constants.

Literals (the fixed value of constants)

- *programiz
- A literal is a value (or an identifier) whose value cannot be altered in a program. For example, `1`, `2.5`, `'c'`.

Integer literals

- Example

  ```
  Decimal (base 10): 0, -9, 22 etc
  Octal (base 8): 021, 077, 033 etc
  Hexadecimal (base 16): 0x7f, 0x2a, 0x521 etc
  ```

Floating-point literals

- Example

```
-2.0
0.0000234
-0.22E-5
```

Character literals

- A **character** literal is created by enclosing a *single* character inside *single* quotation marks. For example, `'a'`, `'m'`, `'F'`, `'2'`, `'}'`.
- Escape sequences

Escape Sequences	Character
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null character

String literals

- A **string** literal is a *sequence* of characters enclosed in *double*-quote marks.

```
"good"                //string constant
""                    //null string constant
"      "              //string constant of six white space
"x"                   //string constant having a single character.
"Earth is round\n"    //prints string with a newline
```

Special Symbols

Symbols		Description
Brackets	`[]`	Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Symbols		Description
Parentheses	`()`	These special symbols are used to indicate function calls and function parameters.
Braces	`{}`	These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.
Comma	`,`	It is used to separate more than one statements like for separating parameters in function calls.
Semicolon	`;`	It is an operator that essentially invokes something called an initialization list.
Asterisk	`*`	It is used to create pointer variable.

assignment operator: It is used to assign values. pre processor |#| The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

Operators

Arithmetic operators

- Operators

Operator	Meaning
`+`	addition or unary plus
`-`	subtraction or unary minus
`*`	multiplication
`/`	division
`%`	remainder after division (modulo division)

Increment and decrement operators

- C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by `1`.

Assignment operators

- An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`.

Operator	Example	Same as
`=`	`a = b`	`a = b`
`+=`	`a += b`	`a = a+b`
`-=`	`a -= b`	`a = a-b`
`*=`	`a *= b`	`a = a*b`

Operator	Example	Same as
/=	a /= b	a = a/b
%=	a %= b	a = a%b
<<=		
>>=		
&=		
^=		
|=		

- Examples: *tutorialspoint

Relation operators

- A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Operator	Meaning	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Logical operators

- Operators

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5)|| (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

Bitwise operators

- During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Operators	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

- Truth table

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

- Examples: *tutorialspoint *tutorialspoint

  Given that

  ```
  A = 0011 1100
  B = 0000 1101
  ```

  ```
  A&B    = 0000 1100
  A|B    = 0011 1101 (turn on the bits of A selected by B)
  A^B    = 0011 0001
  ~A     = 1100 0011
  ~B     = 1111 0010
  A&(~B) = 0011 0000 (turn off the bits of A selected by B)
  ```

  - Turn on bits: *geeksforgeeks
  - Turn off bits: *geeksforgeeks

Operator precedence and associativity

- *geeksforgeeks

Punctuators

- Punctuators: ( ) [ ] { } * , : = ; ... #

- Punctuation characters such as brackets `[ ]`, braces `{ }`, parentheses `( )`, and commas `,` are also tokens. *microsoft
- Some punctuation symbols are also operators. The compiler determines their use from context.
- In a C program, the semicolon `;` is a *statement terminator*. That is, each *individual statement* must be ended with a semicolon. It indicates the end of one logical entity.

Declaration / Definition / Initialization

- *cprogramming *geeksforgeeks
- Summary

	Memory	**Variable**	**Function**
Declaration	not allocated	provides the type and name to the compiler	provides function's name, return type, and parameters to the compiler
Definition	allocated	where and how much storage to create for the variable	actual body of the function
Initialization	-	assign an initial value	-

- *Declaration* can be understood as a *subset* of *definition*.
- A variable or function can be *declared* multiple times but it can be *defined* only once. For example, the following throws an error: *'x': redefinition; multiple initialization*.

```
int x = 1; // defined and initialized.
int x = 1; // error
```

Functions

- A function *declaration* tells the compiler about a function's name, return type, and parameters. A function declaration tells the compiler about a function name and how to call the function.
- A function *definition* provides the actual body of the function.
- The *function prototype* is a declaration of a function. Function declarations are typically organized in a header file.

```
int fcn(); // function declaration (prototype)
```

- The following *defines* and thus *declares* the function `fcn`.

```
int fcn() {
} // function definition
```

- Typical function structure

```c
int fcn(); // function declaration

int main() {
    int i = fcn(); // function call
}

int fcn() {
    return 0;
} // function definition
```

Variables

- A variable *definition* tells the compiler *where and how much storage* to create for the variable.
- A variable *declaration* provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable *definition* has its meaning at the time of compilation only. The compiler needs actual variable definition at the time of linking the program. *tutorialspoint
- To only declare a variable without defining, use `extern` keyword. Of course, the variable must be defined in another source file since variables in C must be defined to use. *geeksforgeeks

```c
extern int x;
```

- Multiple variables can be declared at once.

```c
int x1, x2;
```

Variable initialization

- *geeksforgeeks
- Global and static variables are initialized to `0`.
- `for`-loops requires only declaration of a counter variable. (definition is redundant because a `for`-loop initializes the counter at the beginning.)

Local variables

- Local variables must be initialized before it is used. For instance,

```c
int x;
printf("%d\n", x); // error
```

throws an error in contrast with

```
int x = 2;
printf("%d\n", x); // This prints 2.
```

- Multiple local variable initialization

```
int main() {
    int x1, x2 = 2, arr[] = {0, 1, 2, 3};
    return 0;
}
```

`x1` is only defined, whereas `x2` and `arr` are defined and initialized.

Global variables

- Global variables are automatically initialized to the value `0` during definition if they are not otherwise assigned a value.

```
int x; // `x` is a global variable and initialized to 0.

int main() {
    int y; // `y` is only declared because it is not a global variable, but
a local variable.
    return 0;
}
```

In the above code, `int x;` not only defines `x` but also initialize `x` because `x` is a global variable and hence a storage is allocated to the variable to store a value `0`. (simply speaking, `int x;` includes `x = 0;`.)

- A global variable `int x` defined by `int x;` shares properties of declaration and definition. But it is not equivalent to `int x = 0;` because `int x;` can be written multiple times in a program. That is,

```
int x;
int x; // This is allowed.

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

- Default initial values

Data type	Default initial value
int	`0`
char	`'\0'`
float	`0`
double	`0`
pointer	`NULL`

Preprocessors

- *geeksforgeeks *cprogramming *tutorialspoint *gnu
- In a very basic term, preprocessor takes a C program and produces another C program without any `#`.
- **Directives**: In C and C++, the language supports a simple macro preprocessor. Source lines that should be handled by the preprocessor, such as `#define` and `#include` are referred to as *preprocessor directives*.
- The preprocessor is *not* a statement. That is, the statement terminator `;` is not used.
- List of preprocessors

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

`#include`

- `< >`: standard folder
- `" "`: current folder (directory)

`#define`/`#undef`/`#ifdef`/`#ifndef`/`#endif` (preprocessor macro)

Constants (object-like macros)

- When we use `#define` for a *constant*, the preprocessor produces a C program where the defined constant is searched and matching tokens are *replaced* with the given expression.

```
#define PI 3.141593

int main() {
    printf("%f\n", PI);
    return 0;
}
```

Function-like macros

- *geeksforgeeks
- The arguments are not checked for data type.
- When a macro is used, the input tokens are treated as a set of strings. They are just replaced to the macro token placeholders.

```
#define MULTIPLY(a, b) a*b
#define MULTIPLY2(a, b) (a)*(b)

int main() {
    printf("%d\n", MULTIPLY(1 + 2, 1 + 2)); // This gives 5.
    printf("%d\n", MULTIPLY2(1 + 2, 1 + 2)); // This gives 9.
    return 0;
}
```

  To avoid such problems, enclose the function body with the parentheses as follows:

```
#define MULTIPLY2(a, b) ((a)*(b))
```

- When the arguments do not have decimal points, the output is automatically assigned the `int` type.

```
#define MULTIPLY(a, b) a*b

int main() {
    printf("%d\n", MULTIPLY(1 + 2, 1 + 2)); // This gives 5.
    printf("%f\n", MULTIPLY(1 + 2, 1 + 2)); // This gives 0.000000.
    printf("%f\n", MULTIPLY(1 + 2.0, 1 + 2)); // This gives 5.000000.
    return 0;
}
```

Token pasting operator ##

- The operator ## is used to concatenate the tokens.

```
#define MERGE(a,b) a##b

int main() {
    printf("%d", MERGE(12, 34)); // This gives 1234.
    return 0;
}
```

```
#define PASTER(n) printf ("var" #n " = %d", var##n)

int main(void) {
    int var12 = 1;
    PASTER(12); // This gives var12 = 1.
    return 0;
}
```

where `PASTER(12);` is replaced by

```
printf ("var12 = %d", var12);
```

during preprocessing.

Stringize operator #

- The operator # is used to convert the token passed to macro to a *string literal*.

```
#define STRING(a) #a

int main() {
    printf("%s", STRING(stringLiteral)); // This gives stringLiteral.
    return 0;
}
```

Macro continuation operator \

- The macros can be written in multiple lines using \. The last line doesn't need to have \.

Predefined macros

- There are some standard macros which can be used to print program file (`__FILE__`), Date of compilation (`__DATE__`), Time of compilation (`__TIME__`) and Line Number in C code (`__LINE__`)

Macro	Description
`__DATE__`	The current date as a character literal in "MMM DD YYYY" format.
`__TIME__`	The current time as a character literal in "HH:MM:SS" format.
`__FILE__`	This contains the current filename as a string literal.
`__LINE__`	This contains the current line number as a decimal constant.
`__STDC__`	Defined as 1 when the compiler complies with the ANSI standard.

Undefine directive `#undef`

- A macro can be undefined by `#undef`.

```
#define  FILE_SIZE 24
#undef  FILE_SIZE
#define FILE_SIZE 36
```

Conditional macro directives `#ifdef`/`#ifndef`/`#endif`

- Example

```
#ifndef MESSAGE
#define MESSAGE "Text"
#endif
```

`#if`/`#else`/`#elif`/`#endif` (conditional directives)

- Preprocessors also support *if-else directives* which are typically used for conditional compilation.
- `#if` expression must be an integer constant.

```
#define INDEX 1
int main() {
#if INDEX < 0
    printf("Error.\n"); // This line is not compiled.
#endif
    return  0;
}
```

```

```

```c
int main() {
    float INDEX = 1;
#if INDEX < 0
    printf("Error.\n"); // This line is compiled.
#endif
    return  0;
}
```

`defined` operator

- The `defined` operator is used to verify that a macro is defined. It is usually used when multiple macros have to be verified at once. *auckland

```c
int main() {
#if  defined (MACRO1)  || !defined (MACRO2)
    printf( "Hello!\n" );
#endif
}
```

`#pragma`

- *geeksforgeeks

`#error`

Data Types

- *programiz *geeksforgeeks *tutorialspoint
- The type of a variable determines how much *space* it occupies in storage and how the *bit pattern* stored is *interpreted*.

Classification	
Basic types	Arithmetic types. (a) integer types and (b) floating-point types.
Enumerated types	Arithmetic types. They are used to define variables that can only assign certain discrete integer values throughout the program.
Void type	The type specifier void indicates that no value is available.
Derived types	(a) pointer types, (b) array types, (c) structure types, (d) union types and (e) function types.

Basic types

- The typical size and string specifier of each type on **32 bit gcc compiler** *geeksforgeeks

Type	Size (bytes)	Format Specifier
`char`	1	`%c`
`int`	4	`%d`
`float`	4	`%f`
`double`	8	`%lf`
`short int`	2	`%hd`
`unsigned int`	4	`%u`
`long int`	4	`%ld`
`long long int`	8	`%lld`
`unsigned long int`	4	`%lu`
`unsigned long long int`	8	`%llu`
`signed char`	1	`%c`
`unsigned char`	1	`%c`
`long double`	12	`%Lf`

> In 32-bit and 64-bit operating systems, the Microsoft C compiler maps long double to type double. Therefore, `sizeof(long double)` yields 8 bytes in *MSVC*. *microsoft

- `sizeof dataType` or `sizeof(dataType)` yields the storage size of the object or type in *bytes*.
- The size of `int` is 4 bytes (32 bits). It can take 2^{32} distinct states from `-2^31 (= -2147483648)` to `2^31-1 (= 2147483647)`.
- `float` and `double` are used to hold real numbers. The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.
- `short` and `long` are equivalent to `short int` and `long int`, respectively.
- `signed` and `unsigned` are *type modifiers* that can change the data storage.

```
unsigned int x;
```

  The sizes of `int` and `unsigned int` are both 4 bytes but the values of `unsigned int` range from `0` to `2^31-1`.

- `char` is equivalent to `uint8_t`.

Void types

- `void` is an incomplete type. It means "nothing" or "no type". A variable can have `void` type. Only functions that return nothing can have `void` as its return type.

Types	Description
Function *returns* as void	There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);
Function *arguments* as void	There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, `int rand(void);`
Pointers to void	A pointer of type `void *` represents the address of an object, but not its type. For example, a memory allocation function `void *malloc( size_t size );` returns a pointer to void which can be casted to any data type.

Variables

- *programiz
- A variable is a container (storage area) to hold data.
- A variable is nothing but a name given to a storage area that our programs can manipulate.
- Each variable in C has a specific *type*, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
- Basic variable types

Type	Description
`char`	Typically a single octet(one byte). This is an integer type.
`int`	The most natural size of integer for the machine.
`float`	A *single-precision* floating point value.
`double`	A *double-precision* floating point value.
`void`	Represents the *absence* of type.

- Keyword `char` is used for declaring character type variables.

```
char s = 'a';
```

The size of `char` is 1 byte.

Decision Making

`if-else` statement

- `if-else` ladder template

```
if (x > 1) {
}
else if (y > 0) {
}
else if (a && b) {
}
.
.
else {
}
```

- C programming language assumes any *non-zero* and *non-null* values as *true*, and if it is *either zero or null*, then it is assumed as *false* value.
- When using if...else if..else statements, there are few points to keep in mind: *tutorialspoint
  - An `if` can have zero or one `else`s and it must come after any `else if`s.
  - An `if` can have zero to many `else if`s and they must come before the `else`.
  - Once an `else if` *succeeds*, *none* of the remaining `else if`s or `else`s will be tested.

`switch` statement

- Syntax

```
switch (expression) {
    case constant1:
    // statements
    break;
    case constant2:
    // statements
    break;
    .
    .
    default:
    // default statements
    break;
}
```

- The expression used in a switch statement must have an *integral* or *enumerated* type.
- Template

```
int num = 0;
switch (num) {
case 1:
    printf("switch: 1\n", num);
```

```
        break;
    case 2:
        printf("switch: 2\n", num);
        break;
    default:
        printf("switch: default\n", num);
        break;
    }
```

- Without `break;`, all statements after the matching label are executed.

```
int main() {
    int n = 1;
    switch (n) {
    case 1:
    case 2:
    case 3:
        print
        f("n = %d\n", n); // This line is executed for n = 1, 2, 3.
        break;
    default:
        break;
    }
    return 0;
}
```

- The `default` clause inside the `switch` statement is optional. The `default` case can be used for performing a task when *none* of the cases is true. No `break;` is needed in the default case.

`?:` operator

- Syntax

```
exp1 ? exp2 : exp3;
```

`exp1` is evaluated first. If it is true, the operator gives the evaluation of `exp2` and gives `exp3` otherwise.

Loops

`for` loop

- Syntax

```
for (initializationStatement; testExpression; updateStatement) {
    // block statements
```

```
}
```

- Flow

```
initialization -> test -> code block -> update -> test -> ... -> test ->
exit
```

- Template 1

```
int i;
for (i = 0; i < 4; i++) {
    print("for: %d\n", i);
}
```

- Template 2

```
for (int i = 0; i < 4; i++) {
    printf("%d\n", i);
}
```

In this case, `i` is a local variable, i.e., it cannot be referenced outside the `for` loop.

- In `for` loop argument `i++` and `++i` are equivalent because the counter `i` is incremented after the statements are executed.

`while` loop

- Syntax

```
while (testExpression) {
    // block statements
}
```

- Flow

```
test -> code block -> test -> ... -> test -> exit
```

- Template

```

```

```
unsigned int i = 0;
while (i < 4) {
    printf("while: %d\n", i);
    i++;
}
```

`do-while` loop

- Syntax

```
do {
// statements
}
while (testExpression);
```

- Flow

```
code block -> test -> code block -> ... -> test -> exit
```

This executes the statement first regardless of the *test value*. Only then, the test expression comes into action afterwards.

`break` statement

- The `break` statement *ends the loop immediately* when it is encountered.

```
break;
```

`continue` statement

- The `continue` statement *skips the current iteration* of the loop and continues with the next iteration.

```
continue;
```

- For the `for` loop, `continue` statement causes the conditional test and increment portions of the loop to execute.
- For the `while` and `do...while` loops, `continue` statement causes the program control to pass to the conditional tests.

`goto` statement

- A `goto` statement in C programming provides an unconditional jump from the `goto` to a labeled statement in the same function.

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

- Avoid use of `goto`.

Infinite loop

- `for`

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

- `while`

```
while (1) {
    // statements
}
```

Functions

- A *function declaration* tells the compiler about a function's name, return type, and parameters. A *function definition* provides the actual body of the function.
- A function must be at least declared *above* the lines of the function call.

Function prototype (declaration)

- A function prototype is simply the *declaration* of a function that specifies function's *name*, *parameters* and *return type*. It doesn't contain *function body*.
- Parameter names are not important in function declaration *only their type* is required. *tutorialspoint
- Syntax

```
returnType functionName(type1 argument1, type2 argument2, ...);
// or
returnType functionName(type1, type2);
```

- Example

```
int fcn(int a, int b);
// or
int fcn(int, int);
```

Calling a function

- Syntax

```
functionName(argument1, argument2, ...);
```

Function definition

- A function definition contains a function body as well as the function declaration.
- Syntax

```
returnType functionName(type1 argument1, type2 argument2, ...) {
    //body of the function
} // `;` is an option.
```

Passing arguments

- In programming, *argument* refers to the variable passed to the function.

Formal parameters

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the *formal parameters* of the function. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. *tutorialspoint
- Actual parameters vs. formal parameters *geeksforgeeks *crasseux *csc
  - *Actual parameters (arguments)* are parameters as they appear in function calls.
  - *Formal parameters* are parameters as they appear in function declarations.

Return statement

- Syntax

```
return (expression);
```

- The type of value *returned from the function* and the return type specified in *the function prototype and function definition* must match.

Function types

- When there's no return value, the return type of the function is set to `void`.

Function call types

- *tutorialspoint

Call by value

- This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Call by reference

- This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Recursion

- A function that calls itself is known as a *recursive function*. And, this technique is known as *recursion*.
- In recursion, the body of a function contains a code calling the function.
- The recursion continues until the body of a function does not call itself. That is, the `return` statement does not contain the function evaluation.
- Example

```
int fibonacci(int i) {
    if (i == 0) {
        return 0;
    }
    if (i == 1) {
        return 1;
    }
    return fibonacci(i-1) + fibonacci(i-2);
}
```

Return value convention

- STM32 HAL

OK	`0x00U`
ERROR	`0x01U`

BUSY	`0x02U`
TIMEOUT	`0x03U`

Scope Rules

- *tutorialspoint
- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language:
  - **Local variables**: inside a function or a block.
  - **Global variables**: outside of all functions.
  - **Formal parameters**: in the definition of function parameters.

Local variables

- They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Global variables

- Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

Formal parameters

- Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

Variable initialization

- When a local variable is defined, it is not initialized by the system.
- Global variables are initialized automatically by the system when they are defined.

Storage class

- Every *variable* in C programming has two properties: **type** and **storage class**. Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable. *programiz

- Storage Classes are used to describe the features of a variable/function. These features basically include the *scope*, *visibility* and *life-time* which help us to trace the existence of a particular variable during the runtime of a program. *geeksforgeeks

storage specifier	storage	intial value	scope	life

storage specifier	storage	intial value	scope	life
auto	stack	garbage	within block	end of block
extern	data segment	zero	global multiple files	till end of program
static	data segment	zero	within block	till end of program
register	CPU register	garbage	within block	end of block

Local variable

- The variables declared inside a block are *automatic* or *local variables*. (An automatic variable is a local variable.) The local variables exist only inside the block in which it is declared.
- There is no initial value of an auto variable (garbage).

Global variable

- Variables that are declared outside of all functions are known as *external* or *global* variables. They are accessible from any function inside the program.
- The initial value of a global variable is *zero*.

Register variable

- *tutorialspoint
- The *register* storage class is used to define *local variables* that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the referencing operator & applied to it (as it does not have a memory location).

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

- The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

Static variable

- The static storage class instructs the compiler to keep a *local variable* in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. *tutorialspoint
- The value of a *static variable persists* until the end of the program. Once it is initialized, its value is stored throughout the program even in a new scope. *geeksforgeeks

```
static int var;
```

- The initial value of a global variable is *zero*.
- Static global variables are not allowed to be shared across the source files using `extern` keyword. That is, `static` keyword confines the scope of a global variable to the source file. *stackoverflow

Extern variable

- `extern` keyword is used to use a *global variable* defined in another source file. By using `extern`, a variable shared in multiple source codes needs not be declared in a header file.
- Example
  A variable `int a` is defined only in *source2.c*. Then, *source1.c* can use the variable by declaring `int a` with `extern` in the `main()` block without explicitly declaring `int a` in a common header.
  - source1.c

```
int main() {
    extern int a;
    printf("%d\n", a); // This prints '2'.
    return 0;
}
```

  - source2.c

```
int a = 2;
```

Array

One dimensional array

- Declaration

```
dataType arrayName[arrSize];
```

`arrayName` has arrSize *elements*. The *index* starts from `0` and ends with `arrSize-1`.

- Initialization

```
int arr[5] = {0, 1, 2, 3, 4};
```

```
int arr1[] = {0, 1, 2, 3, 4}; // declaration without specifying the size
int arr2[]; // This throws an error because the length is not specified.
```

- Change value

```
arr[0] = 1;
```

Multidimensional array

- Declaration

```
dataType arrayName[arrSize1][arrSize2]...;
```

- Initialization

```
int arr[2][3] = {{1, 3, 0}, {-1, 5, 9}}; // or
int arr[][3] = {{1, 3, 0}, {-1, 5, 9}}; // or
int arr[2][3] = {1, 3, 0, -1, 5, 9}; // or
int arr2[][2][2] = { {{0, 1}, {2, 3}}, {{4, 5}, {6, 7}} };
```

Initializing an array to the same value

- *geeksforgeeks

```
int arr[ind_1] ... [ind_n] = {};
// or
int arr[ind_1] ... [ind_n] = { 0 };
```

Passing arrays to a function

- To pass arrays to a function, *only the name* of the array is passed to the function. Array names are *decayed to pointers* in the function. *programiz
- C performs no bounds checking for formal parameters.
- One dimensional array: `[]` informs that the array is one-dimensional.

```
int fcn(int arr[]) { // optionally, arr[2]
    // `arr` is a pointer.
    printf("%d\n", arr[0]);
}
```

```c
int main() {
    int arr[] = {0, 1};
    fcn(arr); // This prints 0.
    // Only the name of the array is passed.
    return 0;
}
```

- Multidimensional array: The size of the second or larger index level must be specified.

```c
int fcn(int arr[][2]) { // optionally, arr[2][2]
    // `arr` is a pointer.
    printf("%d\n", arr[1][1]);
}

int main() {
    int arr[][2] = { {0, 1}, {2, 3} };
    fcn(arr); // This prints 3.
    // Only the name of the array is passed.
    return 0;
}
```

    - *geeksforgeeks

Return arrays from a function

- *tutorialspoint

ASCII

- ASCII code table: *genuinecoder
- To find ASCII value of a character,

```c
char chr = 'S';
printf("%d\n", chr);
```

Strings

- A string is *a sequence of characters* terminated with a *null* character `'\0'`. The *string terminator* `'\0'` is automatically added at the end of the string whenever a sequence of characters are enclosed with the *double quotation marks* `""`. Therefore, the length of an array must be *larger* than the number of characters in a string to assign: *programiz

```
char str1[4] = "test"; // incorrect. There is no room for the null `\0`.
char str2[5] = "test"; // correct
char str3[6] = "test"; // correct, the exceeding element is assigned zero.
printf("str1 = %s\n", str1); // This prints `str2` with some garbage strings
added.
printf("str2 = %s\n", str2);
printf("str3 = %s\n", str3);
printf("str2[4] = %d\n", str2[4]); // This prints 0.
printf("str3[4] = %d\n", str3[4]); // This prints 0.
printf("str3[5] = %d\n", str3[5]); // This prints 0.
```

String functions in *string.h*

- *tutorialspoint *programiz

function	
`strcpy(s1, s2);`	Copies string `s2` into string `s1`.
`strcat(s1, s2);`	Concatenates string `s2` onto the end of string `s1`.
`strlen(s1);`	Returns the length of string `s1`.
`strcmp(s1, s2);`	Returns `0` if `s1` and `s2` are the same; less than `0` if `s1<s2`; greater than `0` if `s1>s2`.
`strchr(s1, ch);`	Returns a pointer to the first occurrence of character `ch` in string `s1`.
`strstr(s1, s2);`	Returns a pointer to the first occurrence of string `s2` in string `s1`.

String functions in *stdio.h*

- *programiz
- `fgets()` - takes string input from user
- `puts()` - displays a string

Pointers

Declaration

- Example

```
int* p;
int *p; // preferred
```

```
int * p;
int* p, x; // `p` is a pointer and `x` is an integer.
```

Assigning an address to a pointer

- `&` operator returns the address of the variable.

```
int* p, x = 5; // Valid.
p = &x;
```

```
int x = 5;
int* p = &x; // Valid. This is not `*p = &x;`, but `int *p; p = &x;`.
```

To get the value

- `*` operator is used to get the value of the thing pointed by the pointers.

```
int* p, x, y;
x = 2;
p = &x;
printf("pointer: p = %p, &x = %p\n", p, &x);
printf("value: *p = %d, x = %d\n", *p, x);
```

- `*` is called the *dereference operator* when working with pointers. It operates on a pointer and gives the value stored in that pointer.

To change the value

- Example

```
*p = 3;
```

Pointer to an array

- **Custom notation**: `<==>` indicates *equivalence* and `<=>` means that the two items have the *same value*. These are irrelevant to C syntax.
- `*arr[0]`, `arr[0]`, and `&arr[0]` are equivalent to `**arr`, `*arr`, and `arr`, respectively.
- Multidimensional array indexing: For an `n`-dimensional array `arr`,
    - Value of array

```
arr[ind1][ind2]...[ind_n]
<==> *(*(...*(*(arr+ind1)+ind2)+...)+ind_n) // value of element
```

- Address of the first element of array

```
arr[ind1][ind2]...[ind_k] // k < n
<==> &(*arr[ind1][ind2]...[ind_k])
<==> &arr[ind1][ind2]...[ind_k][0]
```

  - `*arr[ind1][ind2]...[ind_k]` and `arr[ind1][ind2]...[ind_k][0]` are *equivalent*.

- Index level

  `*arr[ind1][ind2]...[ind_k]`, `arr[ind1][ind2]...[ind_k]`, and `&arr[ind1][ind2]...[ind_k]` have the *same value* where `k < n` but they are *not equivalent*. They are different in *index level* where the index levels of the the three pointers are `ind_{k+2}`, `ind_{k+1}`, and `ind_k`, respectively.

```
&arr[l] + n <==> (arr + l) + n <==> &arr[l+n] // line 1
arr[l] + n <==> *(arr + l) + n <==> &arr[l][n] // line 2
*arr[l] + n <==> *(*(arr + l)) + n <==> &arr[l][0][n] // line 3
**arr[l] + n <==> ***(arr + l) + n <==> arr[l][0][0] + n // line 4
// If i!=0, all the three lines represent different values.
```

```
*(arr[l] + m) + n <==> *(*(arr + l) + m) + n <==> *(&arr[l][m]) + n
<==> arr[l][m] + n <==> &arr[l][m][0] + n <==> &arr[l][m][n]
```

- Example: For a three-dimensional array `arr`,

```
arr[1] <==> &arr[1][0] <=> *arr[1] <==> arr[1][0] // These are addresses.
arr[1]+2 <==> &arr[1][2] <=> arr[1][2] // These are addresses.
*(arr[1][2]+1) <=> arr[1][2][1] // Values. identical address, hence value
```

- Memorizing rules

> `&` and the last index `[n]` form a pair that they can be removed together to add an non-negative integer `n`. Equivalently, adding a non-negative integer `n` adds `&` and `[n]`. That is, `&arr[n] <==> arr + n`.
> Removing `*` adds `[n]` at the end of the array indexing: `*(arr + n) <==> arr[n]`.

  - Example

```
*(arr + 1) + 2 <==> arr[1] + 2 <==> &arr[1][2]
```

- Pointer operation
- *geeksforgeeks *stackoverflow *tutorialspoint

Pointers as function arguments (call by reference)

- Example

```
int swap(int* var1, int* var2) {
    int temp = *var1;
    *var1 = *var2;
    *var2 = temp;
    return 0;
}
```

Function pointer

- *geeksforgeeks *cprogramming
- Syntax

```
returnDataType (*fcnPointer) (parameterDataTypes) = &fcn;
```

- Unlike variable pointers, a function pointer *points to code*, not data. Typically a function pointer stores the start of executable code.
- A function's name can also be used to get function's address. Moreover, a function pointer can be used as if it is a function. For example,

```
int fcn(int* p) {
    (*p)++;
    return 0;
}

int main() {
    int x = 0;

    // function pointer definitions
    int (*fcnp1) (int) = &fcn;
    int (*fcnp2) (int) = fcn; // without &

    // function calls: The following four cases are equivalent.
    (*fcnp1)(&x);
    fcnp1(&x); // without *
```

```
    (*fcnp2)(&x);
    fcnp2(&x);

    printf("%d\n", x); // This gives 4.
    return 0;
}
```

- A function pointer can be passed as an argument. For example,

```
int fcn(int* p) {
    (*p)++;
    return 0;
}

int wrapper1(int (*fcnp)(int*), int* p) {
    // function calls: The following two cases are equivalent.
    fcnp(p);
    (*fcnp)(p);
    return 0;
}

// using typedef
typedef int (*FcnType) (int*);

int wrapper2(FcnType fcnp, int* p) {
    // function calls: The following two cases are equivalent.
    fcnp(p);
    (*fcnp)(p);
    return 0;
}

int main() {
    int x = 0;
    int* p = &x;
    int (*fcnp) (int) = fcn;
    // function calls: The following two cases are equivalent.
    wrapper1(fcnp, p);
    wrapper2(fcnp, p);
    printf("%d\n", x); // This gives 4.
    return 0;
}
```

- Function pointers provide a way of passing around *instructions for how to do something*.
- You can write flexible functions and *libraries* that allow the programmer to choose behavior by passing function pointers as arguments.

Subscripting pointer to an array

- *geeksforgeeks

- One-dimensional

```
int *p, arr[]= {...};
p = arr;
```

- Multidimensional

```
arr[][ind_2][ind_3]...[ind_n] = {...};
int (*p)[ind_2][ind_3]...[ind_n];
p = arr;
```

Array of pointers

- "Array of pointers" is an array of the pointer variables. It is also known as pointer arrays. *geeksforgeeks

```
int *p[ind_1]...[ind_n];
```

Combination of pointer to an array and an array of pointers

- Example:

```
int main() {
    int arr0[][2] = { {-1,1},{-2,2} };
    int arr1[][2] = { {-3,3},{-4,4}, {-5, 5} };
    int arr2[][2] = { {-6,6} };
    int(*p[3])[2] = { NULL }; // an array of pointers that have structure of
an array

    p[0] = arr0;
    p[1] = arr1;
    p[2] = arr2;

    printf("%d\n", p[0][0][1]); // This gives 1.
    printf("%d\n", p[1][2][1]); // This gives 5.
    printf("%d\n", p[2][0][0]); // This gives -6.

    printf("%d, %d, %d, %d\n", p, *p, **p, ***p);
    printf("%d, %d, %d, %d\n", p + 1, *(p + 1), **(p + 1), ***(p + 1));
    printf("%d, %d, %d\n", *p + 1, *(*p + 1), **(*p + 1));
    printf("%d, %d\n", **p + 1, *(**p + 1));

    return 0;
}
```

Pointer arithmetic

- *tutorialspoint

Double pointer (pointer to pointer)

- *geeksforgeeks *tutorialspoint
- `*(*p) <==> **p`
- Example:

```
int main() {
    int x = 1;
    int* p;
    int** pp;
    p = &x;
    pp = &p;
    printf("%d, %d\n", *p, **pp); // This gives 1, 1.
    return 0;
}
```

Return pointer from functions

- Syntax

```
int* fcn() {
// statements
}
```

- Example:

```
int* fcn(int* arr1, int* arr2, int size) {
    static int arr[2]; // A local variable to be returned must be `static`.
    for (int n = 0; n < size; n++) {
        arr[n] = arr1[n] + arr2[n];
    }
    return arr;
}

int main() {
    int arr1[] = { -1, -2 }, arr2[] = { 1, 4 }, size = 2;
    int *arr;

    arr = fcn(arr1, arr2, size);
    for (int n = 0; n < size; n++) {
        printf("%d\n", arr[n]);
    } // This gives 0, 2 in order.
```

```
    return 0;
}
```

- Examples: *stackoverflow

Pointer type-casting

- Example: `unsigned int` to `char`

```
int main() {
    unsigned int uiwdata = 0x80;
    printf("%X\n", uiwdata); // This gives 80.
    unsigned char* bdatap = (unsigned char*)&uiwdata;
    for (int n = 0; n < 4; n++) {
        printf("%X\n", *(bdatap + n));
    } // This gives 80, 0, 0, 0 in order.

    printf("%X\n", *(unsigned int*)bdatap); // This gives 80.
    return 0;
}
```

`void` pointers

- *geeksforgeeks *tutorialspoint
- A `void` pointer is a pointer that has no associated data type with it. A `void` pointer *can hold address of any type* and *can be type-casted to any type*.
- It is also called a *general purpose pointer*.
- `*(dataType*) voidPointer` is used to dereference a `void` pointer.
- Example: variable referencing

```
int main() {
    int x = 1;
    void* p = &x;
    printf("%d\n", *(int*)p); // This gives 1.
    printf("%f\n", *(float*)p); // This gives 0.000000.
    printf("%f\n", (float) * (int*)p); // This gives 1.000000.
    printf("%f\n", (float)x); // This gives 1.000000.
    return 0;
}
```

- Example: function argument

```c
int fcn(void* p) {
    *(int*)p = -1;
    return 0;
}

int main() {
    int x = 1;
    void* p = &x;
    fcn(p);
    printf("%d\n", x); // This gives -1.
    return 0;
}
```

- Example: `float` to `char`

```c
typedef union {
    float fdata;
    unsigned char bdata[4];
} c2f; // char 2 float

int main() {
    c2f data;
    data.fdata = 1.2;
    printf("%f\n", data.fdata); // This gives 1.200000.
    for (int n = 0; n < 4; n++) {
    printf("%X\n", data.bdata[n]);
    } // This gives 9A, 99, 99, 3F in order.

    void* p;
    p = &data.bdata;
    printf("%f\n", *(float*)p); // This gives 1.200000.
    return 0;
}
```

- Usage: `qsort` *geeksforgeeks

`NULL` pointers

- It is always a good practice to assign a `NULL` value to a pointer variable in case you *do not have an exact address to be assigned*. This is done at the time of variable *declaration*. A pointer that is assigned `NULL` is called a null pointer.

- In C11 standard

> An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

- `NULL` is a constant defined by

```
#define NULL ((void *)0)
```

- Some of the most common use cases for `NULL` are *geeksforgeeks
    - To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
    - To check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code, e.g., dereference pointer variable *only if* it's not `NULL`.
    - To pass a null pointer to a function argument when we don't want to pass any valid memory address.

Structures

Declaration

- Declaration of a structure

```
struct structureTag {
    type member1; // member declaration
    type member2;
    ...
}; // A storage is not allocated in the declaration.
```

Definition

- Method 1 (data-type-like definition)

```
struct data {
    int x;
    int arr[2];
}; // structure declaration

struct data data1; // global structure variable definition. A storage is
allocated.

int main() {
    struct data data2; // structure variable definition. A storage is
allocated.
    data2.x = 0; // initialization
    data2.arr[0] = 0;
    data2.arr[1] = 0;
    return 0;
}
```

The members of a *global* structure variable are initialized to `0`'s.

- Method 2 (declaration with definition)

```
struct { // The structure tag is optional.
    int x;
    int arr[2];
} data1; // global structure declaration
// The members of data1 are initialized to 0.

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

The *structure tag* is used to define new `struct` variables. If any additional structure variable is not to be defined afterwards, the structure tag can be omitted in the declaration.

- Method 3 (definition with initialization, *designated initialization*, enabled from C99)

```
struct data {
    int x;
    int arr[2];
};

int main() {
    struct data data1 = { 0, {1, 2} }; // definition and initialization
    struct data data2 = { .x = 0, .arr = {1, 2} }; // designated
initialization
    struct data data3 = { .x = -1 }; // designated partial initialization
    return 0;
}
```

The initialization of a structure variable is *not possible* during the declaration.

Member access

- A dot operator `.` is used to access the members of a structure variable.
- A arrow operator `->` is used to access the members of a structure variable through a *pointer*.

Array of structures

- Like other primitive data types, an array of structures can be created.

```
struct data {
    int x;
    int arr[2];
};
```

```
int main() {
    struct data data1[4]; // definition of an array of structures
    data1[0].x = 2;
    data1[0].arr[0] = 3;
    data1[1].x = 4;
    data1[1].arr[1] = 5;
    return 0;
}
```

Pointer of a structure

- Example

```
struct data {
    int x;
    int arr[2];
};

struct data *ptr, data1, data2[2] = { {.x = -1}, {.x = 1, .arr = {2, 3}} };

int main() {
    ptr = &data1;
    printf("data1: %d, %d\n", ptr->x, ptr->arr[0]); // member access
    // This gives data1: 0, 0.
    printf("data2: %d, %d, %d\n", \
        data2->x, (data2 + 1)->arr[0], *((data2 + 1)->arr + 1));
    // This gives data2: -1, 2, 3.
    return 0;
}
```

Bit Fields

- Declaration

```
struct structureTag {
  dataType member1 : width;
  ...
};
```

- *Bit fields* allow the packing of data in a structure.

```
#define TRUE  0x1U
#define FALSE 0x0U

typedef struct __data {
    unsigned int boolean : 1;
    unsigned int bit : 32; // 32 is the maximum bit field size for unsigned
```

```
int.
    int y;
} data;

int main() {
    data x;
    x.boolean = TRUE;
    printf("%d\n", x.boolean); // This gives 1.
    printf("%d\n", sizeof(struct __data)); // This gives 12.
    printf("%d\n", sizeof(data)); // This gives 12.
    return 0;
}
```

- A bit-field member of *bit-field width 1* can be thought of as a *Boolean* variable.
- Array of bit fields is not allowed.
- Pointers are not allowed for bit fields.

`struct` storage class

- Structure members cannot be `static`.

Structure member alignment / padding / data packing

- *geeksforgeeks

Linked list

- *geeksforgeeks *geeksforgeeks

Unions

Declaration

- Declaration of an union

```
struct structureTag {
    type member1; // member declaration
    type member2;
    ...
}; // A storage is not allocated in the declaration.
```

Memory occupation of `union`

- The declaration and definition of `union` are the same as `struct`. The difference between the two is that the whole members of an `union` variable share the same memory address. In case of arrays, the memory address to be shared is of the first element.

- Hence, the bit fields of the `union` members are identical up to the size of each member.
- The memory occupied by a union will be large enough to hold the *largest member* of the union. For example, the memory size of the `union data` is determined by `char str[20];`

```
union data {
    int n;
    float x;
    char str[20];
};

int main() {
    printf("%d\n", sizeof(union data)); // This gives 20.
    return 0;
}
```

Typedef

- `typedef` is used to create an alias name for another data type.

Definition

- Syntax

```
typedef typeDefinition identifier;
```

`typedef` for structures

- Syntax

```
typedef struct __newDataType {
    dataType member1;
    ...
} newDataType;
```

`typedef` for function pointers

- Syntax

```
typedef (*fcnType) (parameterDataTypes) newDataType;
```

- Example:

```
typedef int (*FcnType) (int*);

int fcn(int* p) {
    (*p)++;
    return 0;
}

int main() {
    int x = 0;
    FcnType fcnp1 = fcn;
    int (*fcnp2) (int) = fcn;
    (*fcnp1)(&x);
    (*fcnp2)(&x);
    printf("%d\n", x); // This gives 2.
}
```

`typedef` and `#define`

- `typedef` is limited to giving symbolic names to *types only* where as `#define` can be used to define alias for values as well.
- `typedef` interpretation is performed by the *compiler* whereas `#define` statements are processed by the *preprocessor*.
- `#define` should *not* be terminated with a semicolon `;`, but `typedef` should be terminated with `;`.
- `typedef` follows the *scope rule* which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there. In case of `#define`, when preprocessor encounters `#define`, it replaces all the occurrences, after that (No scope rule is followed). *geeksforgeeks
- `typedef` example

  ```
  typedef char* type;
  type a, b, c;
  ```

  is equivalent to

  ```
  char *a, *b, *c;
  ```

- `#define` example

  ```
  #define TYPE char*
  TYPE a, b, c;
  ```

is equivalent to

```
char* a, b, c;
```

Enumeration

- *geeksforgeeks
- `enum` is a user-defined *data type* in C.

```
typedef enum __ENUM {
    A,
    B
} ENUM;

int main() {
    ENUM e;
    e = A;
    printf("%d\n", e); // This gives 0.
    e = B;
    printf("%d\n", e); // This gives 1.
    return 0;
}
```

- It is mainly used to assign names to integral constants, the names make a program easy to read and maintain. For example,

```
typedef enum __Status {
    STATUS_OK,
    STATUS_ERROR
} Status;

Status fcn(int x) {
    if (x == 0) {
        return STATUS_OK;
    }
    else {
        return STATUS_ERROR;
    }
}

int main() {
    int x = 0;
    if (fcn(x) == STATUS_OK) {
        printf("OK\n"); // This will be executed.
    }
    else {
```

```
        printf("ERROR\n");
    }
    return 0;
}
```

- The members of `enum` are called *enumerators*. The enumerators behave as if they are macros that have a data type.
- The value of enumerators can be assigned by a user.

```
typedef enum __ENUM {
    A = 0x00U,
    B = 0x01U
} ENUM;
```

Type Casting

- Type casting is a way to *convert* a variable from one data type to another data type using the *cast operator* `()`.

```
(dataType) expression
```

Precedence

- *geeksforgeeks
- Example:

```
(double) a / b <==> ((double) a) / b
```

Integer promotion

- Integer promotion is the process by which values of integer type "smaller" than `int` or `unsigned int` are converted either to `int` or `unsigned int`.

```
int main() {
    int n = 17;
    char c = 'c'; // Its ASCII value is 99.
    int sum;
    sum = n + c;
    printf("%d\n", sum); // This gives 116.
}
```

Header

- A header file is a file with extension *.h* which contains C function *declarations* and *macro definitions* to be shared between several source files.
- Including a header file is equal to copying the content of the header file.
- Do not *define* variables or functions in a header file. Do only *declaration*.
- A simple practice in C programs is that we keep all the *constants*, *macros*, *system-wide global variables*, and *function prototypes* in the header files and include that header file wherever it is required.

Header inclusion

`#include` file name syntax

- `< >`: standard folder, system libraries
- `" "`: current folder (directory)

`#include` guard

- A header file may be included more than one time directly or indirectly, this leads to problems of re-declaration of same variables/functions. To avoid this problem, *directives* like `#defined`, `#ifdef`, and `#ifndef` are used. Refer to *wikipedia.

- Example:

```
// header.h
#ifndef __TEST_H
#define __TEST_H

... // header contents

#endif
```

Dynamic Memory Allocation

- *geeksforgeeks *tutorialspoint
- C dynamic memory allocation can be defined as a procedure in which the size of a data structure (like Array) is changed *during the runtime*.
- `calloc()`, `malloc()`, `realloc()`, and `free()` are built-in functions in *stdlib.h* that facilitate dynamic memory allocation.

Functions for dynamic memory allocation

- `void *calloc(int num, int size);` (contiguous allocation)
  This function allocates an array of `num` elements each of which size in bytes will be `size`. It initializes each block with a default value `0`.

```
p = (castType*)calloc(num, elementSize);
```

- `void free(void *address);`
  This function releases a block of memory block specified by address.

- `void *malloc(int num);` (memory allocation)
  This function allocates an array of `num` bytes and leave them uninitialized.

  ```
  p = (castType*)malloc(byteSize);
  ```

  where `byteSize` is often defined by

  ```
  byteSize = elementSize * sizeof(dataType)
  ```

- `void *realloc(void *address, int newSize);`
  This function re-allocates memory extending it up to `newSize`.

  ```
  p = (castType*)realloc(p, newSize);
  ```

  - `realloc` de-allocates the old object pointed to by `p` and returns a pointer to a new object that has the size specified by `newSize`.
  - `p` should be a pointer of dynamically allocated memory.

- If space is insufficient, *allocation fails* and returns a `NULL` pointer.

- String example:

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void maStrcpy(char* text, char* input) {
    if (text == NULL) { // Check if the malloc pointer is NULL.
        printf("allocation failed.\n");
        return 1;
    }
    else {
        strcpy(text, input);
    }
}

int main() {
    char* text;
    text = (char*)malloc(5 * sizeof(char)); // memory allocation
```

```
    maStrcpy(text, "test");
    printf("%s\n", text);

    text = (char*)realloc(text, 10 * sizeof(char)); // memory re-allocation
    maStrcpy(text, "test_text");
    printf("%s\n", text);

    free(text); // memory de-allocation, release

    return 0;
}
```

Initialization

- *geeksforgeeks
- `malloc()` does not initialize the allocated memory. If we try to access the content of memory block then we will get garbage values.
- `calloc()` allocates the memory and also initializes the allocates memory block to `0`.
- Because `malloc()` does not do initialization, it is faster than `calloc()`.

Dynamic allocation of array of structures

- Example:

```
typedef struct __data {
    int x[2];
    double y;
} data;

int main() {
    // contiguous allocation of array of structures.
    data* sarr;
    sarr = (data*)calloc(2, sizeof(data));

    if (sarr == NULL) {
        printf("allocation failed.\n");
        return 1;
    }
    else {
        sarr[0].y = 1.0;
        sarr[1].y = 1.0;
    }

    printf("%d\n", sizeof(data)); // This gives 16.
    printf("%d\n", sizeof(*sarr)); // This gives 16.

    printf("%d\n", sarr[0].x[0]); // This gives 0.
    printf("%lf\n", sarr[0].y); // This gives 1.000000.
    printf("%d\n", sarr[1].x[0]); // This gives 0.
    printf("%lf\n", sarr[1].y); // This gives 1.000000.
```

```
    sarr = (data*)realloc(sarr, 4 * sizeof(data));

    if (sarr == NULL) {
        printf("allocation failed.\n");
        return 1;
    }
    else {
        sarr[3].x[0] = 0;
        sarr[3].y = 1.0;
    }

    printf("%d\n", sarr[3].x[0]); // This gives 0.
    printf("%lf\n", sarr[3].y); // This gives 1.000000.

    free(sarr);

    return 0;
}
```

`NULL` check

- The output of the memory allocation functions should be checked whether it is a `NULL` pointer or not. If it is, the allocation has *failed*.

Dynamic allocation of two-dimensional array

- *geeksforgeeks
- Example: (The 3rd method in *geeksforgeeks)

```
int main() {
    int row = 2, col = 3;
    int **p = (int **)malloc(row*sizeof(int*)); // allocation of array of
pointers.
    for (int n = 0; n < row; n++) {
        p[n] = (int*)malloc(col*sizeof(int)); // allocation of arrays
    }

    p[0][0] = 0;
    p[1][0] = 0;

    for (int n =0; n < row; n++) {
        free(p[n]); // de-allocation of arrays
    }
    free(p); // de-allocation of array of pointers.

    return 0;
}
```

Memory layout of C program

- *geeksforgeeks

C Build (Compilation) Process

- *geeksforgeeks
- 2010B2 - Bryant - Computer Systems A Programmer's Perspective

Phases

- Preprocessing
    - Removal of Comments
    - Expansion of Macros
    - Expansion of the included files
    - Conditional compilation
- Compilation
- Assembly
- Linking

Object Files

Static Library

Shared Library (Dynamic-link Library)

- *geeksforgeeks