A

C Programming Course

Eng. Mohamed Yousef

(Slides Book)

Egypt, 2020

The link of C programming course on my youtube channel is:

https://www.youtube.com/playlist?list=PLfgCIULRQavzxY-IO2sO5Vj5x7C_tjW3R

Follow me on:

https://www.youtube.com/mohamedyousef2

https://electronics010.blogspot.com/

https://www.facebook.com/electronics010

Resources

Books:

- C Programming for the absolute beginner
- Beginning programming with C for dummies
- C _ how to program

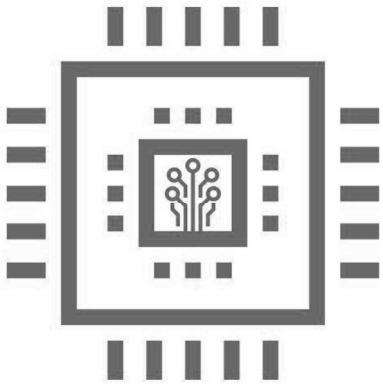
Websites:

- https://www.geeksforgeeks.org/c-programming-language/
- https://www.tutorialspoint.com/cprogramming/
- https://codeforwin.org/2017/08/introduction-c-programming.html
- https://microchipdeveloper.com/tls2101:start

In addition to many scattered web pages.

Embedded Systems

C language



01- Introduction

Eng. Mohamed Yousef

C In an Embedded Environment

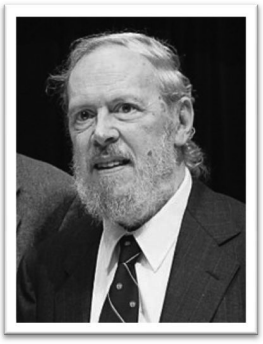
Just the Facts

- C was developed in 1972 in order to write the Unix® operating system
- C is more "low-level" than other high-level languages such as Pascal or Basic
- C is supported by compilers for a wide variety of MCU architectures
- C can do *almost* anything assembly language can do
- C is usually easier and faster for writing code than assembly language

➢ ***C*** was originally developed at Bell Labs by ***Dennis Ritchie*** for use in developing operating systems.

➢ The first major use for C was in writing the ***Unix operating system*** back in 1974.

➢ Due to the low level nature of operating system programming , C has a number of features that are a real benefit to the embedded systems developer.



electronics010.blogspot.com C language Eng. Mohamed Yousef

C -Environment Setup

➢ *Compiler* is a program that checks *source code file* for syntactical errors and translates the source file to low-level *machine code*, if it is error free.

➢ There are many C compilers available such as GCC C, Borland Turbo C, Visual C, Quick Cetc.

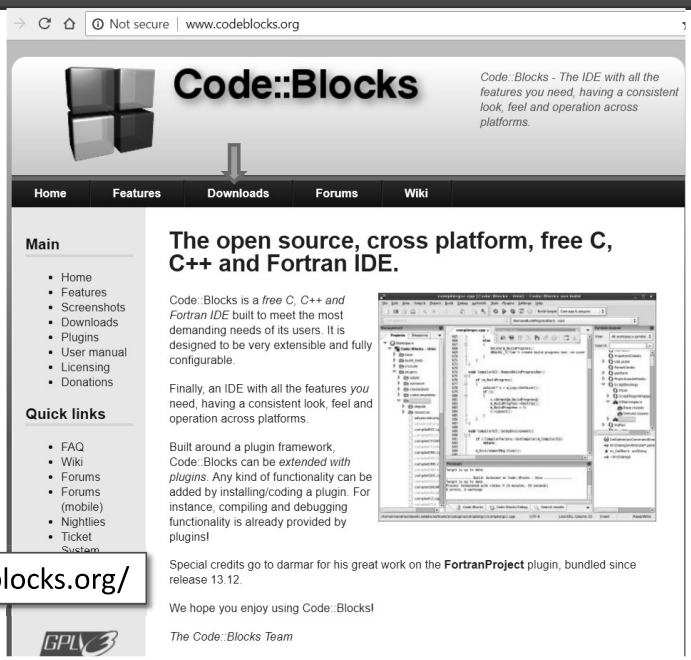
➢ *GCC (GNU Compiler Collection) C* compiler is reliable, efficient and popular C/C++ compiler among developers.

➢ It is open source and available to download for all most all operating systems.

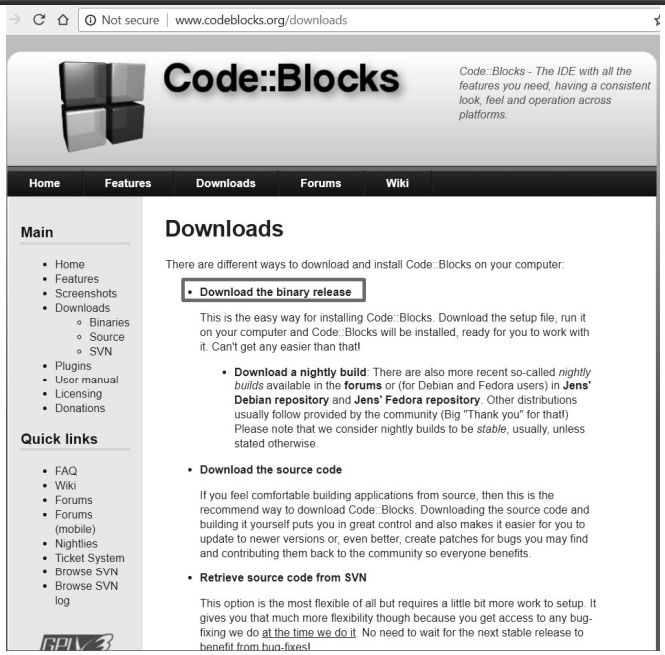
➢ It is pre-installed C compiler in all UNIX based operating systems.

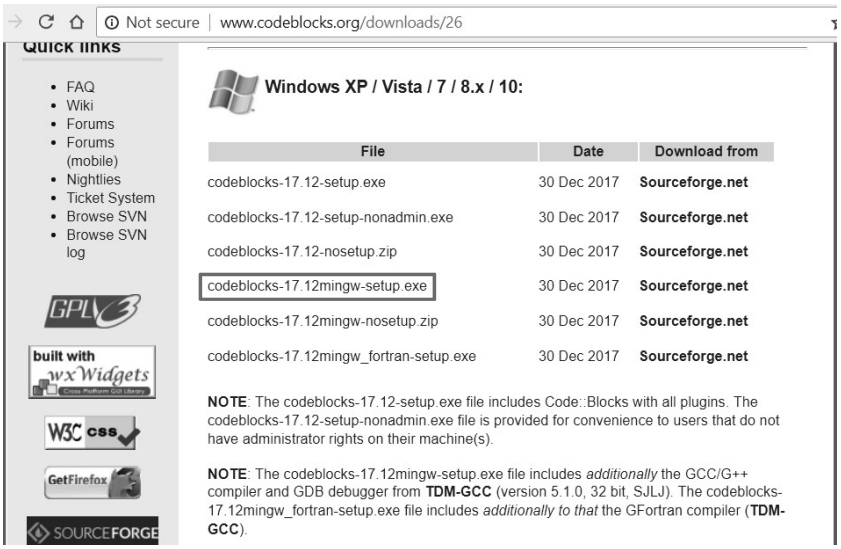
➢ There are many IDE's for developing C projects such as *CodeBlocks, Netbeans, Eclipse, DevC++, Microsoft Visual Studio* etc.

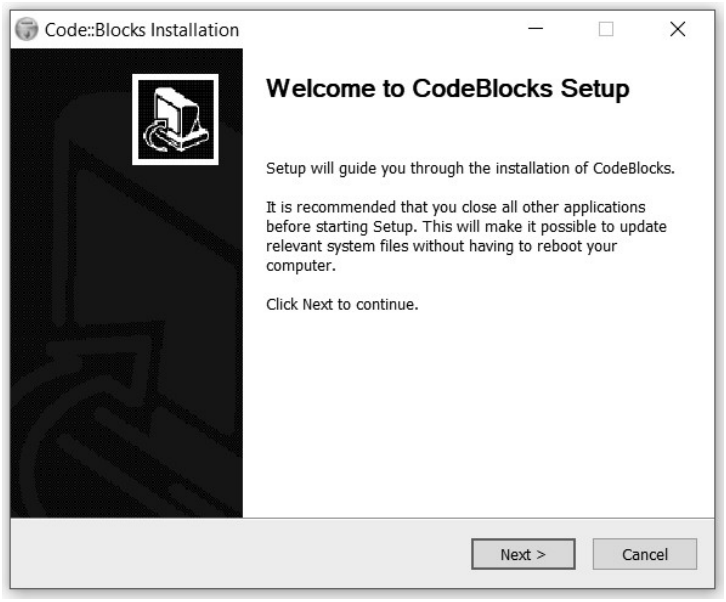
C -Environment Setup



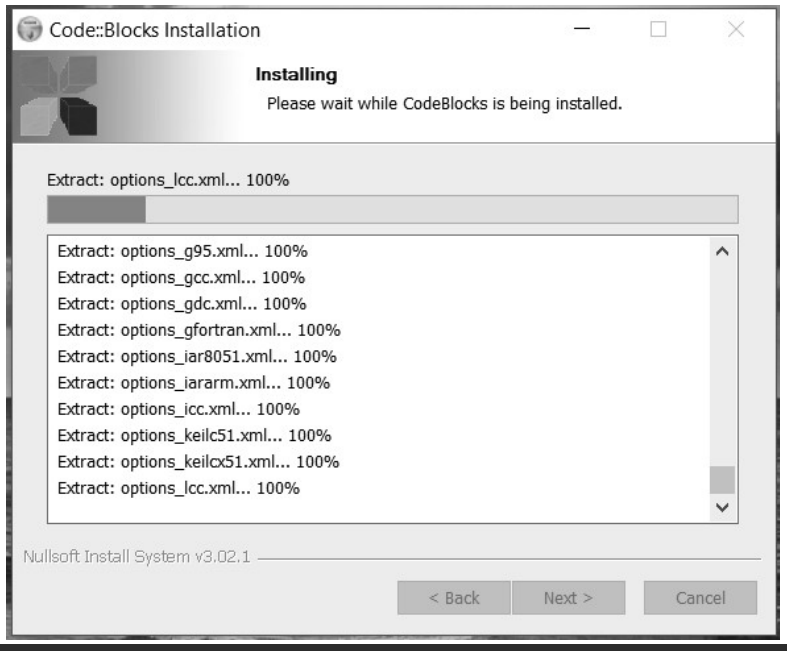
http://www.codeblocks.org/



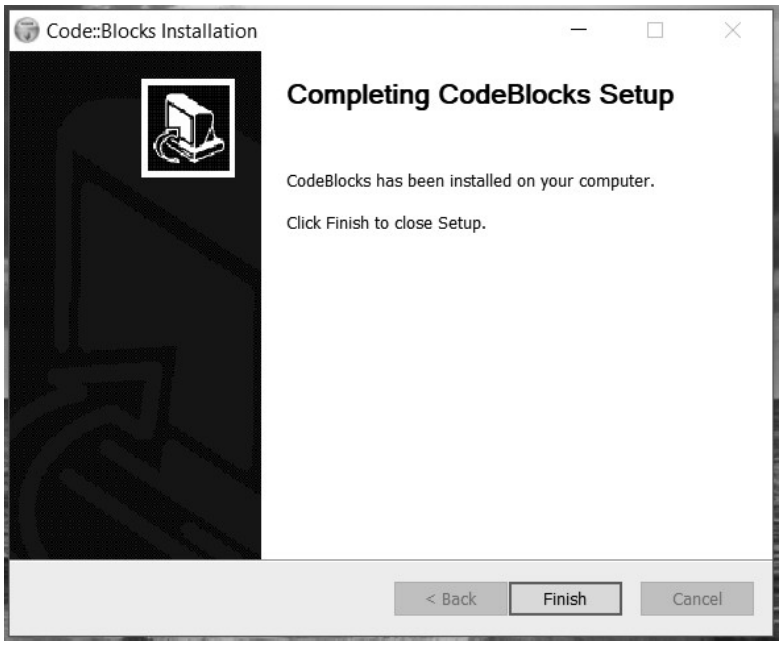




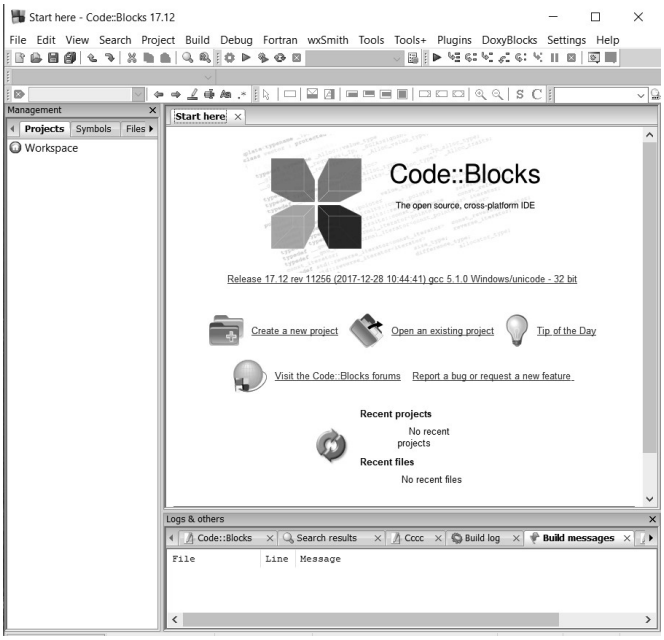
electronics010.blogspot.com C language Eng. Mohamed Yousef



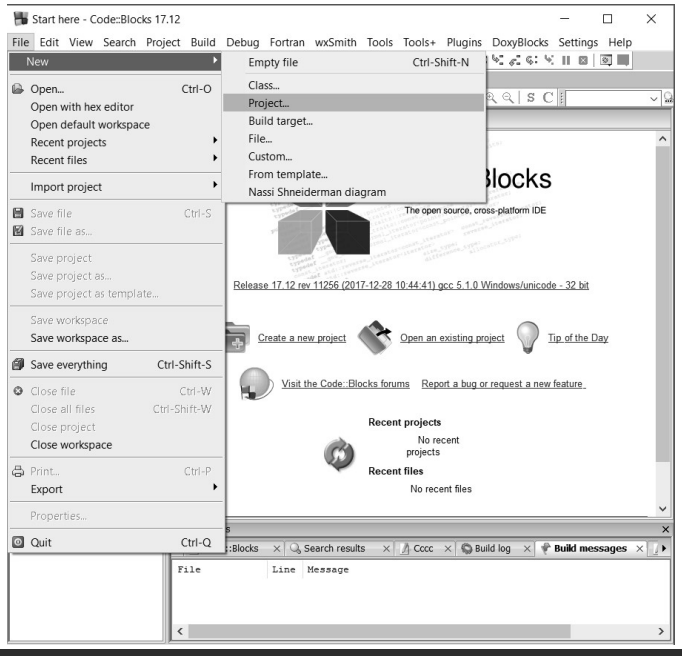
electronics010.blogspot.com C language Eng. Mohamed Yousef



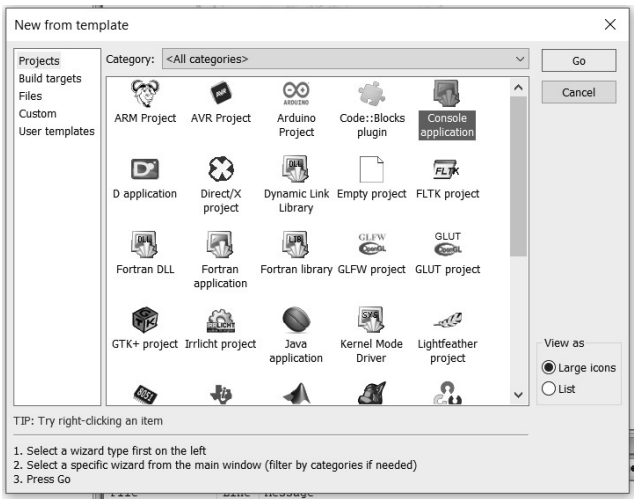
First C Program

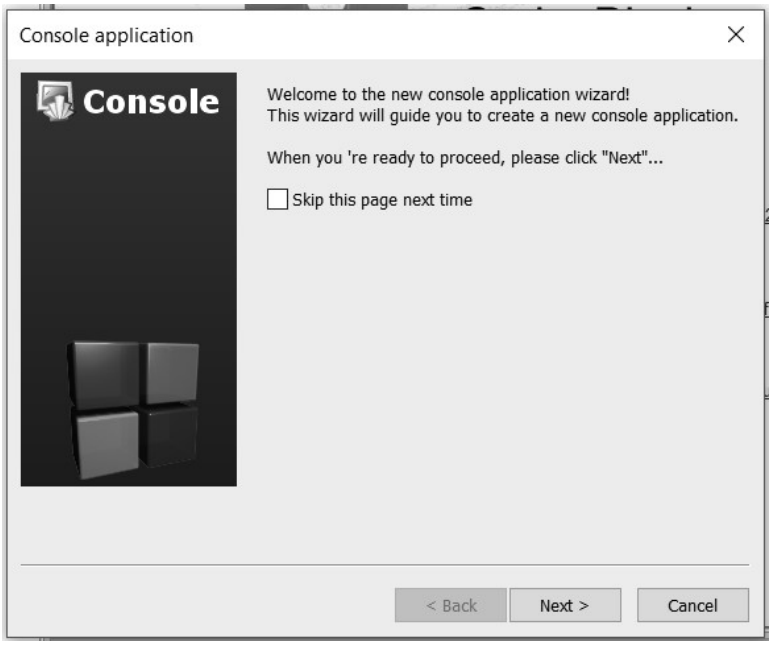


electronics010.blogspot.com C language Eng. Mohamed Yousef

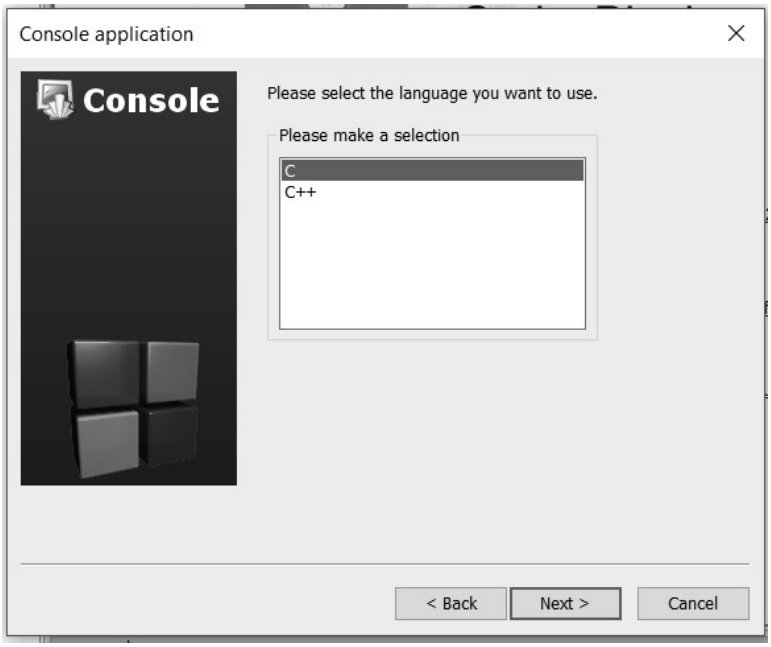


electronics010.blogspot.com C language Eng. Mohamed Yousef

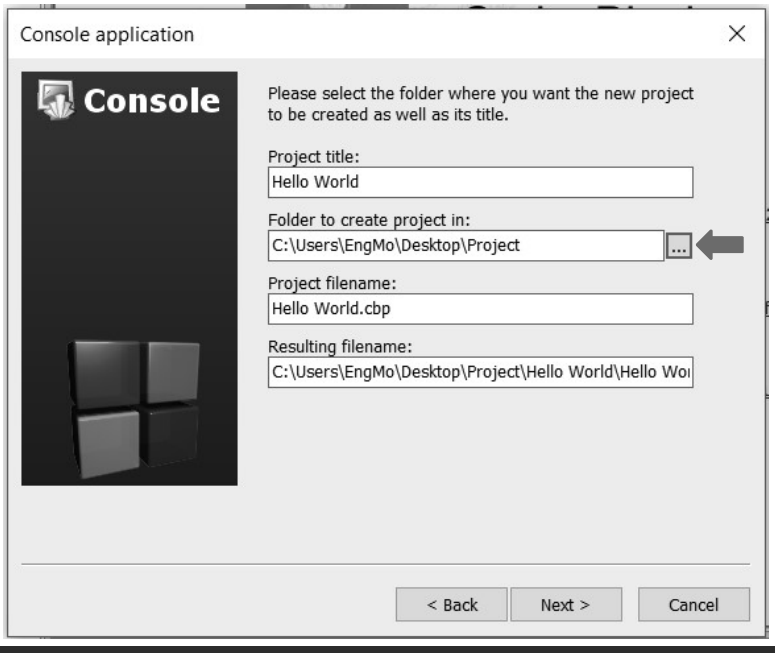


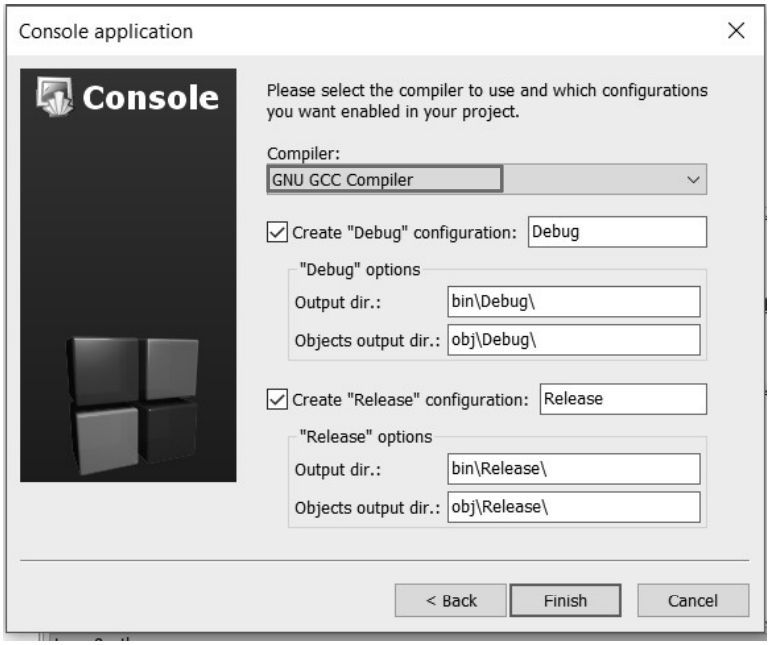


electronics010.blogspot.com C language Eng. Mohamed Yousef

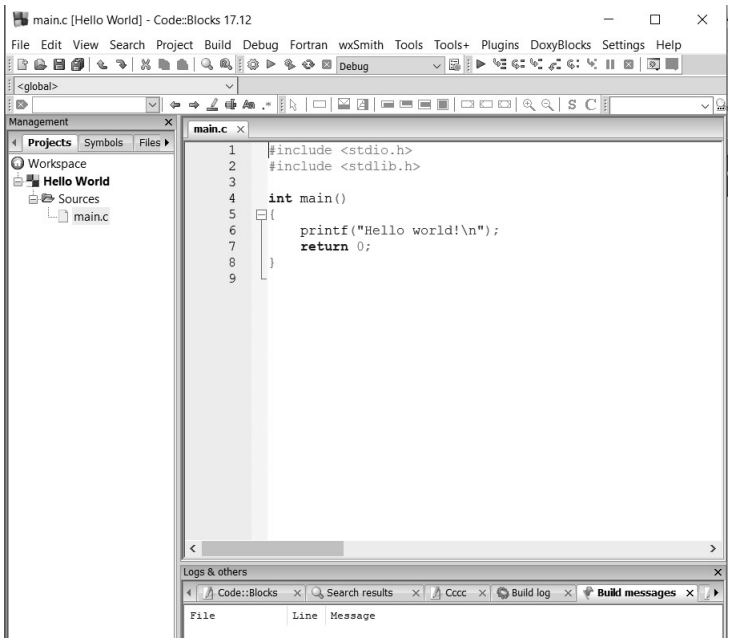


electronics010.blogspot.com C language Eng. Mohamed Yousef

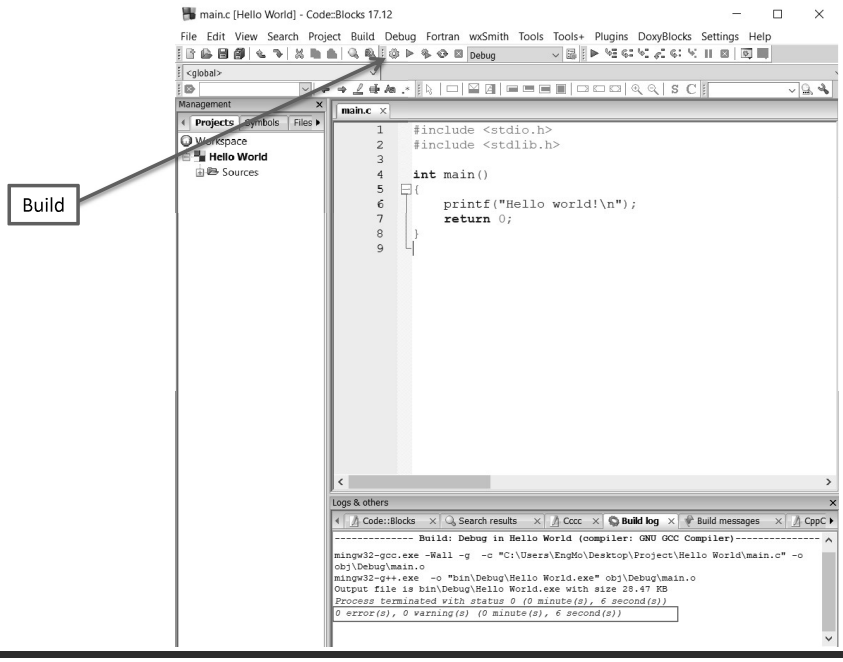


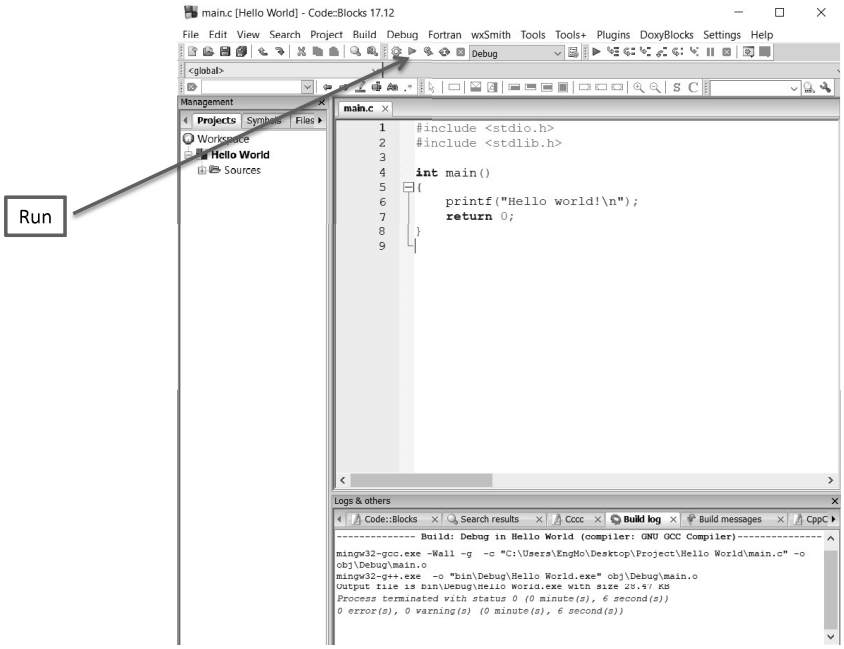


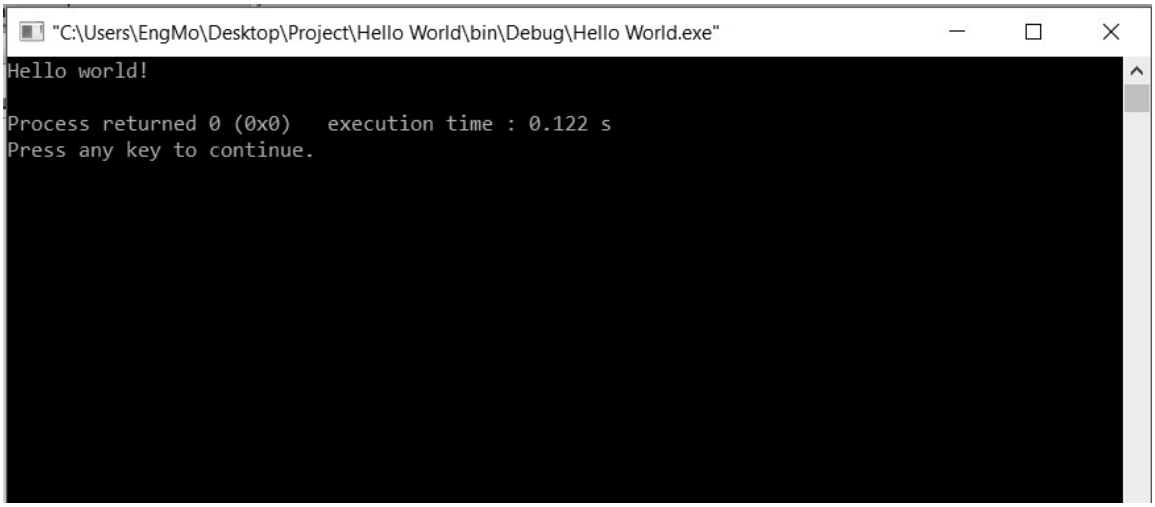
electronics010.blogspot.com C language Eng. Mohamed Yousef



electronics010.blogspot.com C language Eng. Mohamed Yousef







C - Program Structure

Comments

Comments are lines or blocks of text used to document a program's functionality and explain how a program works for the benefit of a programmer. Comments are ignored by the compiler, so you can type anything you want into them.

C supports two different styles of comments: **Block Comments** and **Single Line Comments**.

This indicates the beginning of a comment.

This text is a comment. It doesn't affect the program.

```
/* This is my first program
   in C language
 */

#include <stdio.h>

int main(void)
{
    printf("Hello world!");  // print message
    return 0;
}
```

This indicates the end of a comment.

CAUTION

If your C compiler supports C++, which gcc does, you can use the single line // character set for one-line commenting. Though unlikely, be aware that not all C compilers support the single line character set.

C - Program Structure

Include Directive

Sometimes variables and other program elements are declared in a separate file called a **header file**. Header file names customarily end in **.h**.

1 Header File is in Compiler Path

```
#include <filename.h>
```

C:\Program Files (x86)\CodeBlocks\MinGW\include

2 Header File is in Project Directory

```
#include "filename.h"
```

NOTE
If the preprocessor can not find the file in the place for "" inclusion it will reprocess the directive as if it used <> inclusion syntax.

The **include** directive is used to include the contents of an external file into your program.

The header file **stdio.h** provides standard input/output functions, one of which is **printf()**, which outputs information to the display.

```
/* This is my first program
   in C language
 */

#include <stdio.h>

int main(void)
{
    printf("Hello world!");  // print message
    return 0;
}
```

Preprocessing directives begin with the symbol # and specify actions to be carried out before your program is compiled.

electronics010.blogspot.com C language Eng. Mohamed Yousef

C - Program Structure

main() Function

This is the main() function. Every C program must have one, and only one main() function. This is where your application code resides and is the first thing to run.

This opening brace indicates the start of the body of the **main()** function.

This closing brace indicates the end of the **main()** function.

```
/* This is my first program
   in C language
 */

#include <stdio.h>

int main(void)
{
    printf("Hello world!");  // print message
    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

C - Program Structure

printf() function is used to print the "character, string, float, integer values" onto the output screen.

The **printf()** statement will display the text enclosed in quotes.

```
/* This is my first program
   in C language
 */

#include <stdio.h>

int main(void)
{
    printf("Hello world!");  // print message
    return 0;
}
```

C - Program Structure

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon.

```
/* This is my first program
   in C language
 */

#include <stdio.h>

int main(void)
{
    printf("Hello world!");  // print message
    return 0;
}
```

The body of the function contains executable statements, each of which must be terminated with a semicolon.

electronics010.blogspot.com C language Eng. Mohamed Yousef

C - Program Structure

main() Function

main() function, unlike any other function, will implicitly return a value of 0 upon reaching the } that terminates the function.
(Formerly an explicit return 0; statement was required.)
This is interpreted by the run-time system as an exit code indicating successful execution.

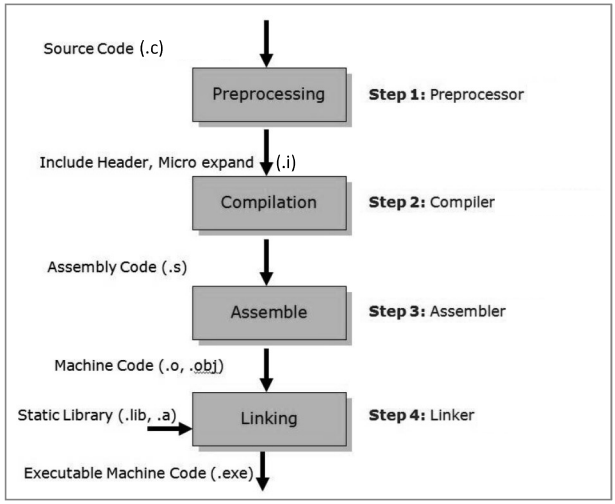
```
/* This is my first program
   in C language
 */

#include <stdio.h>

int main(void)
{
    printf("Hello world!");  // print message
    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Compiling a C program

There are four phases for a C program to become an executable:

- Pre-processing
- Compilation
- Assembly
- Linking



Type of errors in C

Programming errors often remain undetected until the program is compiled or executed. The most common errors can be broadly classified as follows:

1) Syntax errors:

Errors that occur when you violate the rules of writing C syntax are known as syntax errors.

All these errors are detected by compiler and thus are known as compile-time errors.

Most frequent syntax errors are:

- Missing Parenthesis (})
- Printing the value of variable without declaring it
- Missing semicolon like this:

```
// C program to illustrate
// syntax error
#include<stdio.h>
 int  main()
{
    int x = 10;
    int y = 15;

    printf("%d", (x, y)) // semicolon missed
}
```

Error:

```
error: expected ';' before '}' token
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Type of errors in C

2) Run-time Errors:

Errors which occur during program execution (run-time) after successful compilation are called run-time errors.

One of the most common run-time error is division by zero.

These types of error are hard to find as the compiler doesn't point to the line at which the error occurs.

```
// C program to illustrate
// run-time error
#include<stdio.h>
 int  main()
{
    int n = 9, div = 0;

    // wrong logic
    // number is divided by 0,
    // so this program abnormally terminates
    div = n/0;

    printf("resut = %d", div);
}
```

Error:

```
warning: division by zero [-Wdiv-by-zero]
     div = n/0;
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Type of errors in C

3) Logical Errors:

On compilation and execution of a program, desired output is not obtained when certain input values are given.

These types of errors which provide incorrect output but appears to be error free are called logical errors.

These are one of the most common errors done by beginners of programming.

These errors depend on the logical thinking of the programmer and are easy to detect if we follow the line of execution and determine why the program takes that path of execution.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    // logical error : a semicolon after loop
    for(i = 0; i < 3; i++);
    {
        printf("loop ");
    }

    return 0;
}
```

```
loop
Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```

Type of errors in C

4) Semantic errors:

This error occurs when the statements written in the program are not meaningful to the compiler.

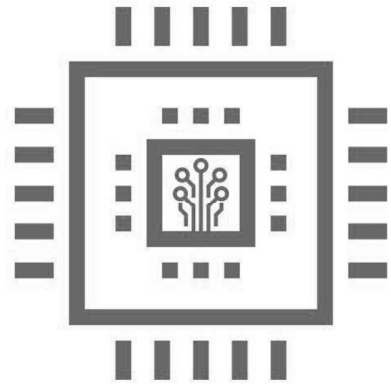
```
// C program to illustrate
// semantic error
 int main()
{
  int a, b, c;
  a + b = c; //semantic error
}
```

Error

```
error: lvalue required as left operand of assignment
a + b = c; //semantic error
```

Embedded Systems

C language



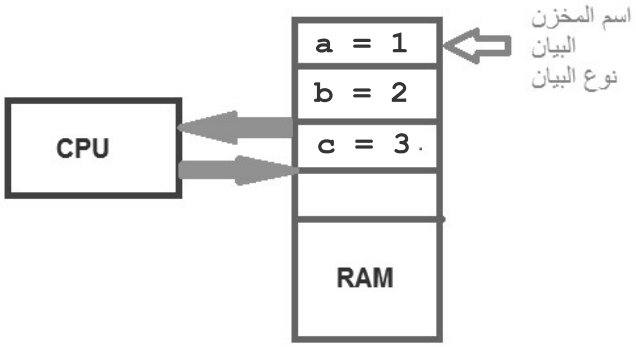
02- Primary Data Types

Eng. Mohamed Yousef

Variables

لكى يتم تنفيذ برنامج تمت كتابته فإنه يستخدم الذاكرة RAM لوضع البيانات ونتائج التنفيذ بها، ولكى يتم التعامل مع الأماكن التى تحتوى على البيانات فى الذاكرة فإنه يتم تسميتها, وأيضا تحديد نوع البيان الذى سيتم تخزينه فى هذا المكان.

```
1 + 2 = 3
a = 1
b = 2
c = a + b = 3
```



type name = value

electronics010.blogspot.com C language Eng. Mohamed Yousef

Variables

الشروط الواجب توافرها عند التسمية:

- يتكون الإسم من الحروف A → Z كبيرة أو صغيرة ، وأيضا الأرقام 0 → 9 ، والعلامة " _ " underscore فقط.
- يجب أن يبدأ الإسم بحرف أو علامة " _ " ولايجب أن يبدأ برقم.
- لا يجب أن يتخلل الأسم مسافات.
- لا يزيد الاسم عن 31 حرف.
- لا يجب أن يكون الإسم من الكلمات المحجوزة أو الأوامر الخاصة بلغة C .

Reserved words

C89 has 32 reserved words, also known as keywords, which are the words that cannot be used for any purposes other than those for which they are predefined:

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

C99 reserved five more words:

`_Bool`	`_Imaginary`	`restrict`
`_Complex`	`inline`	

C11 reserved seven more words:[25]

`_Alignas`	`_Atomic`	`_Noreturn`	`_Thread_local`
`_Alignof`	`_Generic`	`_Static_assert`	

electronics010.blogspot.com C language Eng. Mohamed Yousef

Case sensitivity

Here case sensitive means whether the letters are capital or not.

Keywords remain case sensitive and they must be written in lower case.

C variables are case sensitive

```
total    TOTAL    Total    ToTal    total    totaL
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Variables

والأمثلة التالية توضح مسميات صحيحة

```
temperature_degree
MyName
SUM_4
_ID02
```

Variables

بينما الأمثلة التالية توضح مسميات خاطئة

```
your  address                 'يتخلله مسافة
4you                          ' يبدأ برقم
book.color                    '  يوجد به حرف خاص وهو النقطة
```


Types

يوجد بعض الأنواع الأساسية التى تستخدم مع تحديد نوع المتغيرات وهى:

char – تستخدم لتخزين أحد رموز Ascii، كقيمة حرفية، أو لتخزين رقم صحيح.

ASCII TABLE American Standard Code for Information Interchange

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[END OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

الأسكي، كغيرها من أنظمة تمثيل الرموز في الحاسوب، تحدد العلاقة بين قيمة رقمية لتتابع البتات وبين رمز أو رسم مستخدم في اللغة المكتوبة.

Types

يوجد بعض الأنواع الأساسية التى تستخدم مع تحديد نوع المتغيرات وهى:

- char – تستخدم لتخزين أحد رموز Ascii، كقيمة حرفية، أو لتخزين رقم صحيح.
- int – تستخدم لتخزين الأرقام الصحيحة.
- float – تستخدم لتخزين الأرقام العشرية.
- double – تستخدم لتخزين الأرقام العشرية بدقة أعلى أو قيمة اكبر.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Types

كما تستخدم كلمات للدلالة على كون الرقم موجب أم موجب و سالب

- signed – للدلالة على أن المتغير يمكن أن يحتوى على قيمة رقمية سالبة او موجبة.
- unsigned– للدلالة على ان المتغير يجب أن يحتوى على قيمة رقمية موجبة فقط.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Types

كما تستخدم كلمات لتحديد أكبر قيمة رقمية يمكن تخزينها وهى

- short
- long

Types

The following table provides the details of standard types with their storage sizes and value ranges -

Type	Size(Byte)	Minimum	Maximum
[signed] char	1	-128	127
unsigned char	1	0	255
[signed] short [int]	2	-32768	32767
unsigned short [int]	2	0	65535
[signed] int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
[signed] long [int]	4	-2147483648	2147483647
unsigned long [int]	4	0	4294967295
[signed] long long [int]	8	-9223372036854775808	9223372036854775807
unsigned long long [int]	8	0	18446744073709551615
float	4	1.175494E-038	3.402823E+038
Precision value: 6			
double	8	2.225074E-308	1.797693E+308
Precision value: 15			
long double	12	3.362103E-4932	1.189731E+4932
Precision value: 18			

electronics010.blogspot.com C language Eng. Mohamed Yousef

أمثلة

```
unsigned short int a;
a = 1;

unsigned short int a = 1;
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

أمثلة

```
unsigned short int a;
unsigned short int b;

unsigned short int a, b;
a = 1;
b = 2;
a = b = 3;
```

أمثلة

```
unsigned short int a = 1, b = 2;
```


أمثلة

```
unsigned char count, initial;
count = 120;
initial = 'T';
initial = 84;
```


أمثلة

```
1  unsigned int a =100;    // right
2  unsigned int a=-100;    // wrong
```

أمثلة

```
int a=10;   // right
int a=-10;  // right
signed int a=10;    // right
signed int a=-10;   // right
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Variable Declarations and Definitions

- A *declaration* describes the name and type of identifier .
- A *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.
- No space is reserved in memory for any variable in case of declaration.

- A *definition* tells the compiler where and how much storage to create for the variable in memory.

DEFINITION = DECLARATION + SPACE RESERVATION

electronics010.blogspot.com C language Eng. Mohamed Yousef

Variable Declarations and Definitions

Most of the times, variable declaration and definition are done together.

```
int x;
int y = 1;
```

Variable Declarations and Definitions

Is it possible to have separate declaration and definition?

It is possible in case of *extern* variables and *functions*.

Constants

- It is a fixed value that the program may not change.
- It can have any of the data type.
- There are two simple ways in C to define constants :
  - Using #define preprocessor.
  - Using const keyword.

Constants

The #define Preprocessor:

```
#define label text
```

```
#define OCTO 8
```

```
#define AUTHOR "Dan Gookin"
```

```
#define CELLS 24*80
```

Because this is a preprocessor, everything this line does is finished before the program is passed on to the compiler.
The preprocessor will replace every instance of label with text.
The primary benefit in using #define text substitutions for constants is that they have no impact on memory usage.

Constants

The const Keyword:

```
const type identifier = value;
```

```
const int x = 20;
const float PI = 3.14;
```

This has the effect of creating a variable, allocating memory for it, and initializing it with the value, but the const keyword will prevent you from building any code that changes the value of the variable.

While this is fine on a PC where memory is cheap, it is a very expensive proposition in the embedded world.

It takes up as much space as any other variable.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Literal Constants

literal constant

A **literal constant** or simply a **literal** is a value, such as a number, character, or string that may be assigned to a variable or symbolic constant, used as an operand in an arithmetic or logical operation, or as a parameter to a function.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Integer Literal Constants

The following statements are identical:

```
i = 42;                 // decimal integer
i = 0x2a;               // hexadecimal integer → (0x, 0X)
i = 052;                // octal integer
i = 0b101010;           // binary integer → (0b, 0B)
```

Literal Qualifiers

Much like variables, literals may be qualified to force the compiler to treat them as a specific data type.

Qualifiers are specified by adding a suffix to the value.

- `U` or `u` for `unsigned`: `25u`
- `L` (preferred) or `l` for `long`: `25L`
- `F` or `f` for `float`: `10.25f`

The `U` and `L` suffixes may be combined to create an `unsigned long` literal:

Note: Order of the qualifiers doesn't matter means `ul` or `lu` both are same.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Literal Qualifiers

In C programming a float or real constant is specified either using decimal or using exponential notation.

Decimal notation

Valid examples of float constants in decimal notation.

`1.2` `-0.4` `0.` `.3` `-.3`

Exponential notation

`7850000000000` → `7.85e12`

`0.0000000000785` → `7.85e-011`

Valid examples of real constants in exponential notation

`0.1e1` `0e-5` `5e25` `+5E+10` `-5.2e0`

electronics010.blogspot.com C language Eng. Mohamed Yousef

Character Literal Constants

Character literal is a single character constant enclosed within single quotes.
There are many ways to represent a character constant in C programming.

1. Using character inside single quote. For example, `'a'` `'0'` `'/'` `'.'` etc.
2. Using escape sequence characters. For example, `'\n'`
3. Using an integer ASCII representing of a character. For example, `65` in ASCII format is used to represent `'A'`.
4. Using octal or hexadecimal representation of integer as escape sequence character. For example, `\05` `\xA`

Character Literal Constants

List of escape sequence characters

Escape character	Description
\0	NULL
\a	Alert (Beep)
\n	New line
\r	Carriage return
\t	Horizontal Tab (Eight blank spaces)
\\	Backslash
\'	Single quotes
\"	Double quotes

Examples:

```
#define New_Line '\n'
#define Carriage_Return '\r'
```

```
#define New_Line 10
#define Carriage_Return 13
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

electronics010.blogspot.com C language Eng. Mohamed Yousef

Write 2 separate C programs to declare the next constants by using `#define` and `const` keywords .

NUM_1 = -3,000,000,000

NUM_2 = 12.3456

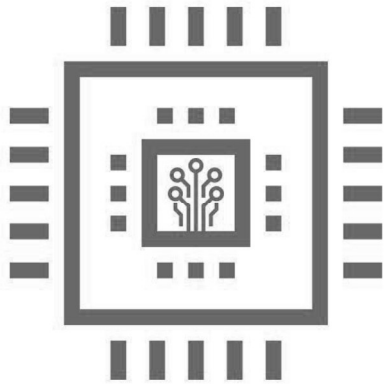
FLAG = 'T'

```
/*
================================================================================
==
PROGRAM     : Lab01-1
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/

#define NUMB_1  -3000000000
#define NUMB_2  12.3456
#define FLAG    'T'

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

```
/*
================================================================================
==
PROGRAM     : Lab01-2
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/

const signed long long int NUMB_1 = -3000000000;
const float NUMB_2 = 12.3456;
const char FLAG = 'T';

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

Embedded Systems

C language



03- Inputs and Outputs

Eng. Mohamed Yousef

The printf() Function

The *printf()* function sends a formatted stream of text to the standard output device.
The name *printf()* means print formatted.

```
printf("text");
```

```
printf("Hello World!");
```

```
Hello World!
Process returned 0 (0x0)   execution time : 0.173 s
Press any key to continue.
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

Employing escape sequences

To reference certain characters that you cannot type into your source code, the C language uses an escape sequence.

```
printf("Hello World!\n");
```

```
Hello World!

Process returned 0 (0x0)   execution time : 0.082 s
Press any key to continue.
```

Escape character	Description
\n	New line
\r	Carriage return
\t	Horizontal Tab (Eight blank spaces)
\\	Backslash
\"	Double quotes

The printf() Function

Employing escape sequences

To reference certain characters that you cannot type into your source code, the C language uses an escape sequence.

```
printf("Hello World!\n");
```

```
Hello World!

Process returned 0 (0x0)   execution time : 0.082 s
Press any key to continue.
```

Escape character	Description
\n	New line
\r	Carriage return
\t	Horizontal Tab (Eight blank spaces)
\\	Backslash
\"	Double quotes

electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

Employing escape sequences

To reference certain characters that you cannot type into your source code, the C language uses an escape sequence.

```
printf("Double quote character \"");
```

```
Double quote character "
Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
```

Escape character	Description
\n	New line
\r	Carriage return
\t	Horizontal Tab (Eight blank spaces)
\\	Backslash
\"	Double quotes

electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

Employing escape sequences

To reference certain characters that you cannot type into your source code, the C language uses an escape sequence.

```
printf("Double quote character \"\n");
```

```
Double quote character "

Process returned 0 (0x0)   execution time : 0.131 s
Press any key to continue.
```

Escape character	Description
\n	New line
\r	Carriage return
\t	Horizontal Tab (Eight blank spaces)
\\	Backslash
\"	Double quotes

The printf() Function

Printing Variable Contents

```
int x = -35;

printf("%d", x);
```

```
-35
Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

Specifier & Output
c Character
d or i Signed decimal integer
e Scientific notation (mantissa/exponent) using e character (double)
E Scientific notation (mantissa/exponent) using E character (double)
f Decimal floating point (double)

Specifier & Output
g Uses the shorter of %e or %f
u Unsigned decimal integer

electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

Length & Description
h The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, u,
l The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, u
L The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g

%lli, %lld	Signed Integer	long long
%llu	Unsigned Integer	unsigned long long

The printf() Function

```c
#include <stdio.h>

int main()
{
    double x = 3.1416, y = 1.2e-5, z = 5000000000.0;

    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("z = %f\n", z);

    printf("x = %e\n", x);
    printf("y = %e\n", y);
    printf("z = %e\n", z);

    printf("x = %g\n", x);
    printf("y = %g\n", y);
    printf("z = %g\n", z);


    return 0;
}
```

```
x = 3.141600
y = 0.000012
z = 5000000000.000000
x = 3.141600e+000
y = 1.200000e-005
z = 5.000000e+009
x = 3.1416
y = 1.2e-005
z = 5e+009
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

```c
#include <stdio.h>
#define printf __mingw_printf

int main()
{
    long double x = 3.1416, y = 1.2e-5, z = 5000000000.0;

    printf("x = %Lf\n", x);
    printf("y = %Lf\n", y);
    printf("z = %Lf\n", z);

    printf("x = %Le\n", x);
    printf("y = %Le\n", y);
    printf("z = %Le\n", z);

    printf("x = %Lg\n", x);
    printf("y = %Lg\n", y);
    printf("z = %Lg\n", z);


    return 0;
}
```

```
x = 3.141600
y = 0.000012
z = 5000000000.000000
x = 3.141600e+000
y = 1.200000e-005
z = 5.000000e+009
x = 3.1416
y = 1.2e-005
z = 5e+009
```

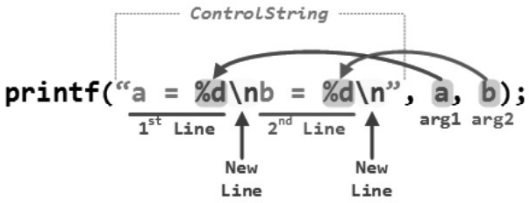
electronics010.blogspot.com C language Eng. Mohamed Yousef

The printf() Function

```c
printf("%f", 3.123456);
printf("\n%.3f", 3.123456);
printf("\n%.4f", 3.123456);
```

```
3.123456
3.123
3.1235
```

```c
{
    int a = 5;
    int b = 10;
    printf("a = %d\nb = %d\n", a, b);
}
```



The code above would produce the following output:

```
a = 5
b = 10
_
```


The scanf() Function

The `scanf()` function is another built in function provided by the standard input output library `<stdio.h>`; it reads standard input from the keyboard and stores it in previously declared variables. It takes two arguments as demonstrated next.

```
scanf("placeholder",variable);
```

In this version, *placeholder* is a conversion character, and *variable* is a type of variable that matches the conversion character. Unless it's a string (`char` array), the variable is prefixed by the `&` operator.

```
scanf("%f",&temperature);
```



```c
#include <stdio.h>

int main()
{
    float num1 = 0.0, num2 = 0.0;

    printf("\t\tAdder Program\n");
    printf("\t\t=============\n");

    printf(" Enter 1st number : ");
    scanf("%f", &num1);

    printf(" Enter 2nd number : ");
    scanf("%f", &num2);

    printf(" %g + %g = %g \n", num1, num2, num1 + num2);


    return 0;
}
```

```
               Adder Program
               =============
 Enter 1st number : 5
 Enter 2nd number : 2.6
 5 + 2.6 = 7.6

Process returned 0 (0x0)   execution time : 17.619 s
Press any key to continue.
```

The scanf() Function

"%f" is the format for a double.

There is no format for a float, because if you attempt to pass a float to printf, it'll be promoted to double before printf receives it.

Note that this is one place that printf format strings differ substantially from scanf format strings.

For input you're passing a pointer, which is not promoted, so you have to tell scanf whether you want to read a float or a double, so for scanf, %f means you want to read a float and %lf means you want to read a double .

Note that scanf not reading long double.

MinGW, which uses the gcc compiler and the Microsoft runtime library. Unfortunately, those components disagree on the underlying type to be used for long double. Windows assumes long double is the same size as double; gcc makes long double bigger.

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

int main()
{

    double num1 = 0.0, num2 = 0.0;

    printf("\t\tAdder Program\n");
    printf("\t\t=============\n");

    printf(" Enter 1st number : ");
    scanf("%lf", &num1);

    printf(" Enter 2nd number : ");
    scanf("%lf", &num2);

    printf(" %g + %g = %g \n", num1, num2, num1 + num2);


    return 0;
}
```

```
                Adder Program
                =============
Enter 1st number : -0.09
Enter 2nd number : 2
-0.09 + 2 = 1.91
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

Exercises

Following program has many syntax errors. Enumerate the errors then re-write the program after correction.

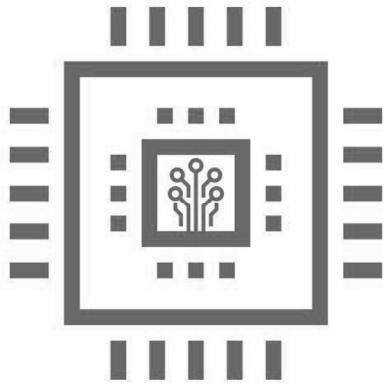
```
#include stdio.h

Void Main(
{
      float w
      printf "Enter your weight in pounds ";
      scanf["%d" weight];
      printf('Your weight in kilograms is %d',
            weight * 0.45359237);
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

int main()
{
    float weight;
    printf ("Enter your weight in pounds ");
    scanf("%f", &weight);
    printf("Your weight in kilograms is %f", weight * 0.45359237);

    return 0;
}
```

Embedded Systems

C language



04- Arithmetic in C

Eng. Mohamed Yousef

Arithmetic Operators

Arithmetic operators are used in arithmetic computations.

Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder (integer division)
++	Auto increment
−−	Auto decrement

Important Note: There is no exponential operator in C programming language. Many beginners considers `^` (hat symbol) as an exponential operator. However, `^` is a bitwise XOR operator.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arithmetic Operators

Integer division

In computer programming divisions are performed differently. Apparently there are two types of division.

1. Integer division
2. Real division

C performs integer division if both operands are integer type. Integer division always evaluates to an integer discarding the fractional part.

C performs real division if any of the two operands are real type (either `float` or `double`). Real division evaluates to a real value containing integer as well as fractional part.

Operation	Result	Division type
`5 / 2`	2	Integer division
`5.0 / 2`	2.5	Real division
`5 / 2.0`	2.5	Real division
`5.0 / 2.0`	2.5	Real division

Arithmetic Operators

Modulo division

Modulo division evaluates remainder after performing division of two numbers. In C programming, we perform modulo division using `%` operator. Many texts refers modulo operator as modulus operator.

Important note:

- Modulo division is only possible with integer operands. It causes compilation error with float types.
- Modulo operator returns sign of the first operand.

Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder (integer division)
++	Auto increment
−−	Auto decrement

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

int main(void)
{
    printf("\n  7 modulus  3 = %d",  7%3);
    printf("\n -7 modulus  3 = %d", -7%3);
    printf("\n  7 modulus -3 = %d",  7%-3);
    printf("\n -7 modulus -3 = %d", -7%-3);
    printf("\n  7 modulus  7 = %d",  7%7);

    return 0;
}
```

```
 7 modulus  3 = 1
-7 modulus  3 = -1
 7 modulus -3 = 1
-7 modulus -3 = -1
 7 modulus  7 = 0
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arithmetic Operators

```c
/* Post-increment operator */

j = 4;
k = j++;        // k = 4, j = 5

/* Pre-increment operator */

j = 4;
k = ++j;       // k = 5, j = 5

/* Post-decrement operator */

j = 12;
k = j--;     // k = 12, j = 11

/* Pre-decrement operator */

j = 12;
k = --j;    // k = 11, j = 11
```

Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder (integer division)
++	Auto increment
−−	Auto decrement

Implicit Type Conversion

In many expressions, the type of one operand will be temporarily "promoted" to the larger type of the other operand.

```
int x = 10;
float y = 2.55, z;

z = x * y;            //x promoted to float
printf("%f", z);
```

```
25.500000
```

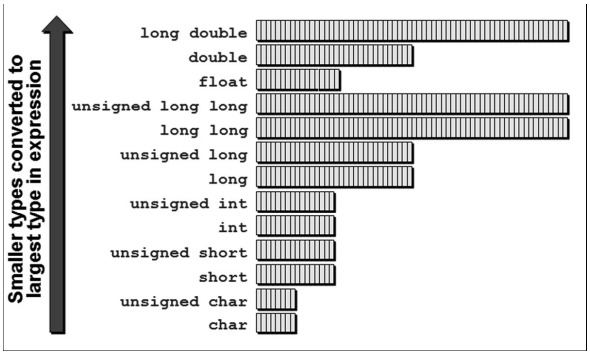

Type Conversion

Type casting is a way to convert a variable from one data type to another data type.

There are two types of type conversion:

1) *Implicit Type Conversion* Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.
- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).




```
int x = 10;
float y = 2.55, z;

z = x * y;            //x promoted to float
printf("%f", z);
```


Type Conversion

At first look, the expression (a*b)/c seems to cause arithmetic *overflow* because signed characters can have values only from -128 to 127 (in most of the C compilers), and the value of subexpression '(a*b)' is 1200 which is greater than 128.

But *integer promotion* happens here in arithmetic done on char types and we get the appropriate result without any overflow.

```
char a = 30, b = 40, c = 10;
char d = (a * b) / c;
printf ("%d ", d);
```

Output:

```
120
```

Type Conversion

Type casting is a way to convert a variable from one data type to another data type.

There are two types of type conversion:

2) Explicit Type Conversion

- This process is also called *type casting* and it is user defined.
- Here the user can type cast the result to make it of a particular data type.

The syntax in C:

```
(type_name) expression
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Type Conversion

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

c = a/b;
printf("%d/%d = %.2f\n",a,b,c);
```

```
3/2 = 1.00
```

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

c = (float)a/(float)b;
printf("%d/%d = %.2f\n",a,b,c);
```

```
3/2 = 1.50
```

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

c = (float)a/b;
printf("%d/%d = %.2f\n",a,b,c);
```

```
3/2 = 1.50
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Operator precedence

```
F = (5 - 1) * (10 - 5);

F = 5 – 1 * 10 – 5;
```

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level", they are evaluated left to right.
*	Multiplication	Evaluated Second. Left to right if there are several.
/	Division	
%	Remainder	
+	Addition	Evaluated Third. Left to right if there are several.
-	Subtraction	
=	Assigment	Evaluated last.

LAB

Write the expected output of the following program. (Justify your answer?)

```c
#include <stdio.h>

int main()
{
    int x = 15, y = 2, z;
    float r;

    z = x/y;
    printf("x/y = %d\n", z);

    r = x/y;
    printf("x/y = %f\n", r);

    r = x/(float)y;
    printf("x/y = %f\n", r);

    r = x/(y * 1.0);
    printf("x/y = %f\n", r);

    return 0;
}
```

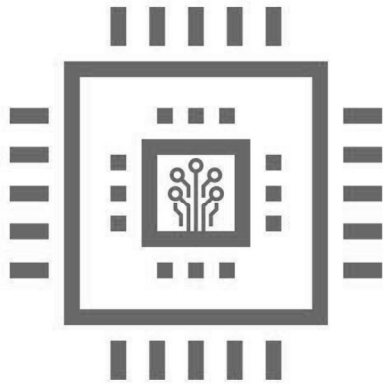

Write the expected output of the following program. (Justify your answer?)

```c
#include <stdio.h>

int main()
{
    int x = 15, y = 2, z;
    float r;

    z = x/y;
    printf("x/y = %d\n", z);

    r = x/y;
    printf("x/y = %f\n", r);

    r = x/(float)y;
    printf("x/y = %f\n", r);

    r = x/(y * 1.0);
    printf("x/y = %f\n", r);

    return 0;
}
```

```
x/y = 7
x/y = 7.000000
x/y = 7.500000
x/y = 7.500000
```

Embedded Systems

C language



05- Decision Making

Eng. Mohamed Yousef

Relational Operators

The relational operators are used to compare two values and tell you whether the comparison being made is *true* or *false*.

```
x = 10
x > 8       // returns true
x == 10     // returns true
x < 100     // returns true
x > 20      // returns false
x != 10     // returns false
x >= 10     // returns true
x <= 10     // returns true
```

Operator	Operation
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

electronics010.blogspot.com C language Eng. Mohamed Yousef

Logical Operators

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration.

Table 8-2 Logical Comparison Operators

Operator	*Name*	*True When*
&&	and	Both comparisons are true
||	or	Either comparison is true
!	not	The item is false

Logical Operators

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration.

```
x = 7;
x > 0 && x < 10     // true
x >= 0 && x < 5     // false
```

Table 8-2 Logical Comparison Operators

Operator	*Name*	*True When*
&&	and	Both comparisons are true
||	or	Either comparison is true
!	not	The item is false

TABLE 3.3 TRUTH TABLE FOR THE AND OPERATOR

x	y	Result
true	true	true
true	false	false
false	true	false
false	false	false

electronics010.blogspot.com C language Eng. Mohamed Yousef

Logical Operators

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration.

```
x = 7;
x > 0 || x < 10     // true
x >= 0 || x < 5     // true
```

Table 8-2 Logical Comparison Operators

Operator	*Name*	*True When*
&&	and	Both comparisons are true
||	or	Either comparison is true
!	not	The item is false

TABLE 3.4 TRUTH TABLE FOR THE OR OPERATOR

x	y	Result
true	true	true
true	false	true
false	true	true
false	false	false

electronics010.blogspot.com C language Eng. Mohamed Yousef

Logical Operators

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration.

```
x = 7;
!(x == 7)     // false
!(x != 7)    // true
```

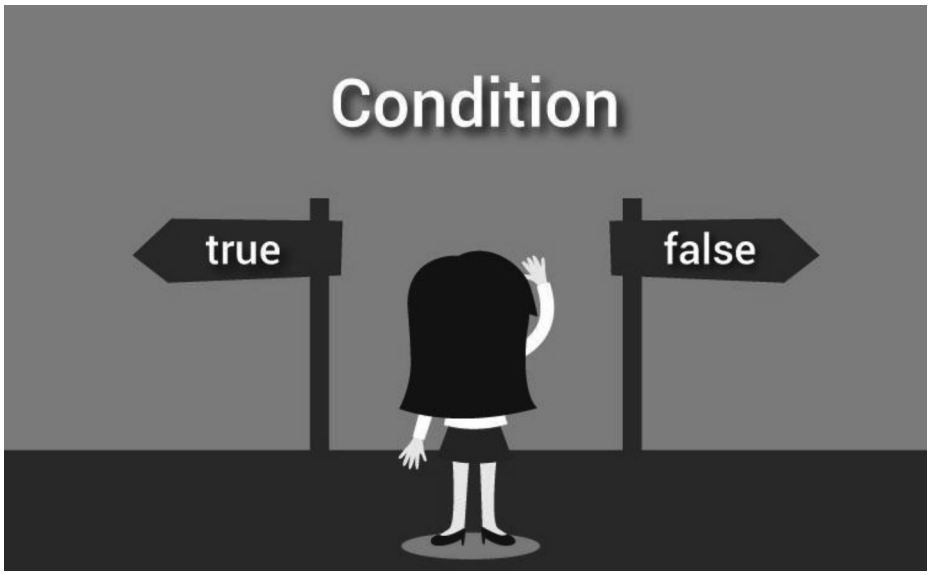
Table 8-2 Logical Comparison Operators

Operator	*Name*	*True When*
&&	and	Both comparisons are true
||	or	Either comparison is true
!	not	The item is false

TABLE 3.5 TRUTH TABLE FOR THE NOT OPERATOR

x	Result
true	false
false	true

Decision Making



Decision Making

- Computer executes programs step by step starting from the program entry point "main" up to the end.
- However sometimes it is required to *change the sequence* of the execution. This called a *Decision making*.
- Sometimes for certain *condition* some steps must be *executed/skipped*. This called a *Branch*.
- Sometimes it is required to *repeat* some steps for a number of times or until certain condition. This called a *Loop*.

C language provides several statements to change the execution flow, those statements are:

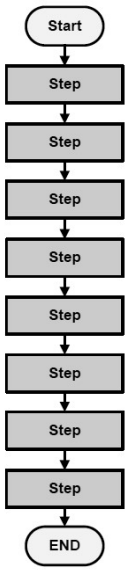
(Branching)

- *if statement*
- *switch statement*

(Looping)

- *for statement*
- *while statement*
- *do…while statement*
- *goto statement c*

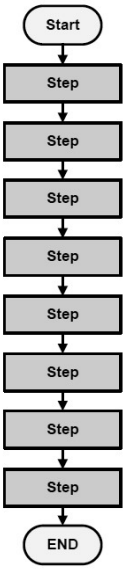
Program Execution Flow



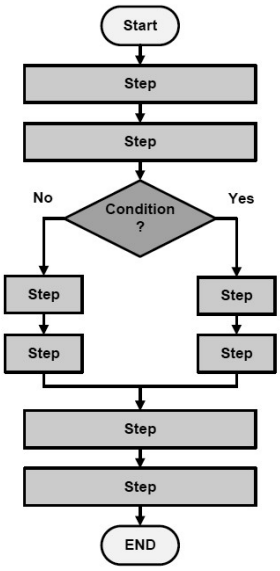
No Loops

No Branch

Program Execution Flow



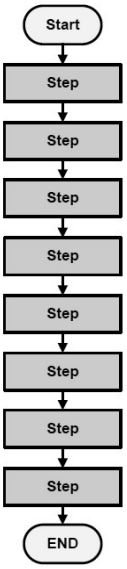
No Loops
No Branch



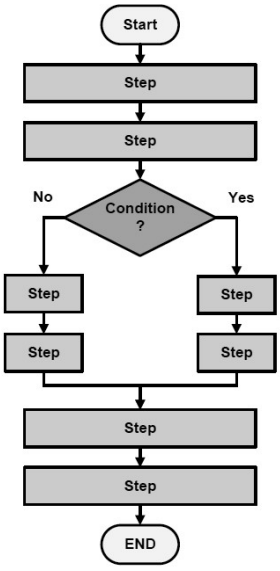
Branch

electronics010.blogspot.com C language Eng. Mohamed Yousef

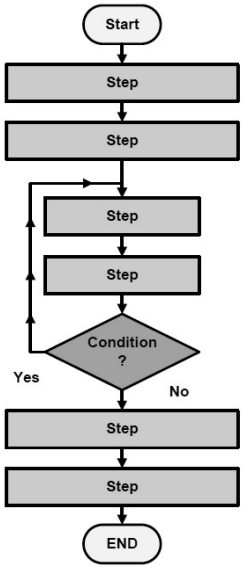
Program Execution Flow



No Loops
No Branch



Branch



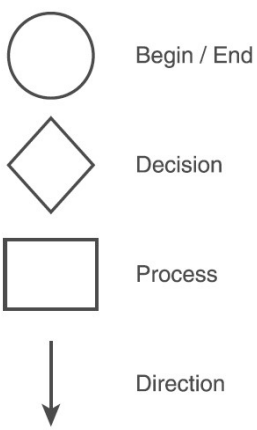
Loop

electronics010.blogspot.com C language Eng. Mohamed Yousef

Flowcharts

Flowcharts use graphical symbols to depict an algorithm or program flow.

Common Flowchart Symbols



Introducing the *if* keyword

The *if* keyword is used to make decisions in your code based upon simple comparisons.

```
if(evaluation)
{
    statement;
}
```

- The evaluation is a *comparison*, a *mathematical operation*, the *result of a function* or some other condition.
- If the condition is *true*, the statements (or statement) enclosed in braces are *executed*; otherwise, they're skipped.

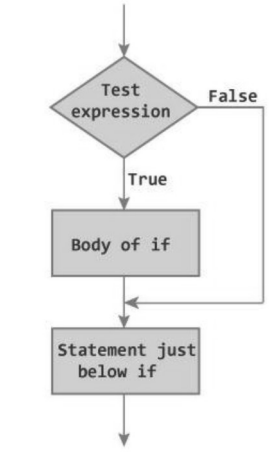


Figure: Flowchart of if Statement

electronics010.blogspot.com C language Eng. Mohamed Yousef

Introducing the *if* keyword

The *if* keyword is used to make decisions in your code based upon simple comparisons.

```
if(evaluation)
{
    statement;
}
```

- The evaluation is a *comparison*, a *mathematical operation*, the *result of a function* or some other condition.
- If the condition is *true*, the statements (or statement) enclosed in braces are *executed*; otherwise, they're skipped.

- The `if` statement's evaluation need not be mathematical. It can simply be a function that returns a true or false value; for example:

```
if(ready())
```

This statement evaluates the return of the `ready()` function. If the function returns a true value, the statements belonging to `if` are run.

- Any non-zero value is considered true in C. Zero is considered false. So this statement is always true:

```
if(1)
```

And this statement is always false:

```
if(0)
```

- When only one statement belongs to an `if` comparison, the braces are optional.

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
#include <stdio.h>

int main(void)
{
    int first,second;

    printf("Input the first value: ");
    scanf("%d",&first);

    printf("Input the second value: ");
    scanf("%d",&second);

    if(first<second)
    {
        printf("%d is less than %d\n",first,second);
    }

    if(first>second)
    {
        printf("%d is greater than %d\n",first,second);
    }

    return 0;
}
```

```
Input the first value: 6
Input the second value: 8
6 is less than 8
```

```c
#include <stdio.h>
#define SECRET 17

int main(void)
{
    int guess;

    printf("Can you guess the secret number: ");
    scanf("%d",&guess);

    if(guess == SECRET)
    printf("You guessed it!");

    if(guess != SECRET) printf("Wrong!");

    return 0;
}
```

```
Can you guess the secret number: 8
Wrong!
```

```
Can you guess the secret number: 17
You guessed it!
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Forgetting where to put the semicolon

```c
#include <stdio.h>

int main(void)
{
    int a = 5, b = -3;

    if(a == b);
    printf("%d equals %d\n",a,b);

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Forgetting where to put the semicolon

```c
#include <stdio.h>

int main(void)
{
    int a = 5, b = -3;

    if(a == b);
    printf("%d equals %d\n",a,b);

    return 0;
}
```

```
5 equals -3
```

Knowing the difference between = and ==

```
#include <stdio.h>

int main(void)
{
    int a = 5;

    if(a = -3){
        printf("%d equals %d\n",a,-3);
    }

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Knowing the difference between = and ==

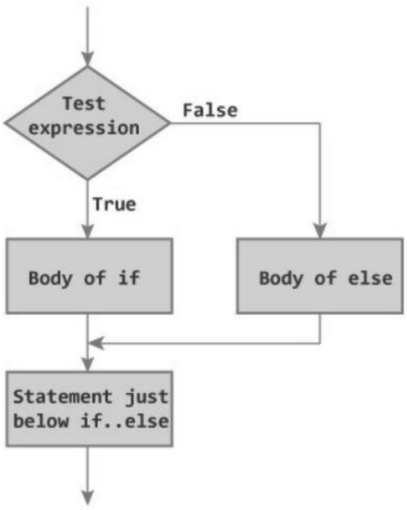
```
#include <stdio.h>

int main(void)
{
    int a = 5;

    if(a = -3){
        printf("%d equals %d\n",a,-3);
    }

    return 0;
}
```

```
-3 equals -3
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Making more-complex decisions

```
if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```



Making more-complex decisions

```
if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

```
#include <stdio.h>

int main(void)
{
    int a = 6, b = a - 2;

    if(a > b)
    {
        printf("%d is greater than %d\n",a,b);
    }
    else
    {
        printf("%d is not greater than %d\n",a,b);
    }

    return 0;
}
```

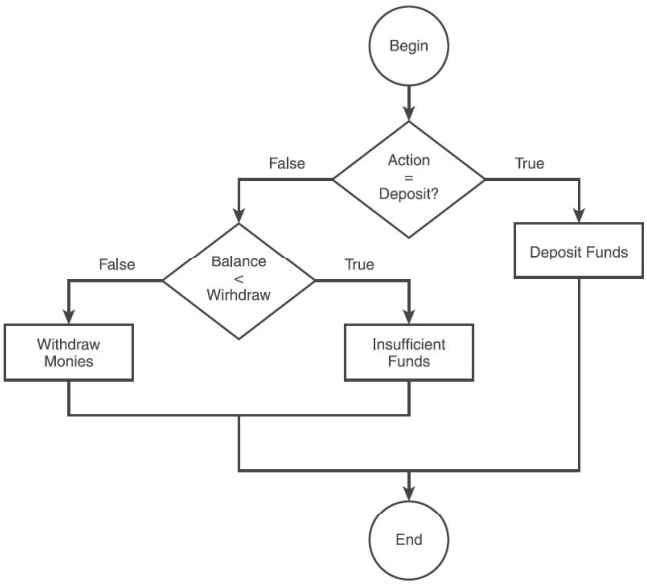
```
6 is greater than 4
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Adding a third option

```
if(condition)
{
    statement(s);
}
else if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

C has no limit on how many else if statements you can add to an if decision process.

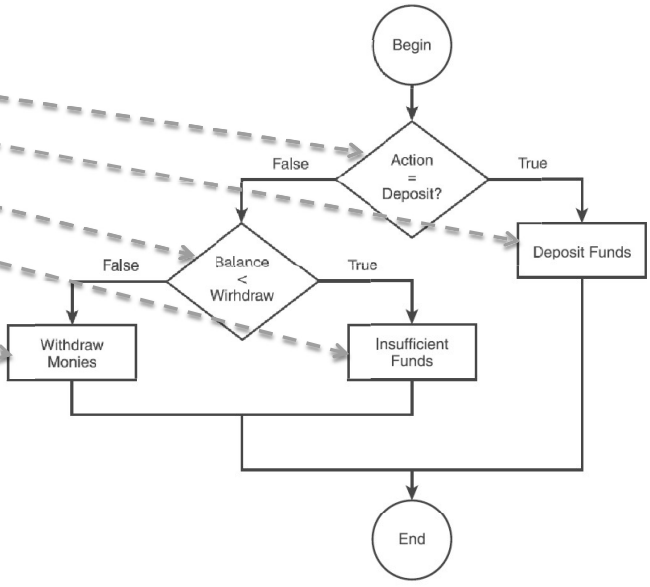


electronics010.blogspot.com C language Eng. Mohamed Yousef

Adding a third option

```
if(condition)
{
    statement(s);
}
else if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

C has no limit on how many else if statements you can add to an if decision process.



Adding a third option

```
if(condition)
{
    statement(s);
}
else if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

C has no limit on how many else if statements you can add to an if decision process.

```
#include <stdio.h>

int main(void)
{
    int x = 9;

    if(x > 5)
    {
        printf("%d is greater than 5\n",x);
    }
    else if(x > 3)
    {
        printf("%d is greater than 3\n",x);
    }
    else
    {
        printf(" x =  %d\n",x);
    }

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Adding a third option

if(condition)
{
    statement(s);
}
else if(condition)
{
    statement(s);
}
else
{
    statement(s);
}

C has no limit on how many else if statements you can add to an if decision process.

#include <stdio.h>

int main(void)
{
    int x = 9;

    if(x > 5)
    {
        printf("%d is greater than 5\n",x);
    }
    else if(x > 3)
    {
        printf("%d is greater than 3\n",x);
    }
    else
    {
        printf(" x = %d\n",x);
    }

    return 0;
}

```
9 is greater than 5
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

#include <stdio.h>

int main(void)
{
    int x = 9;

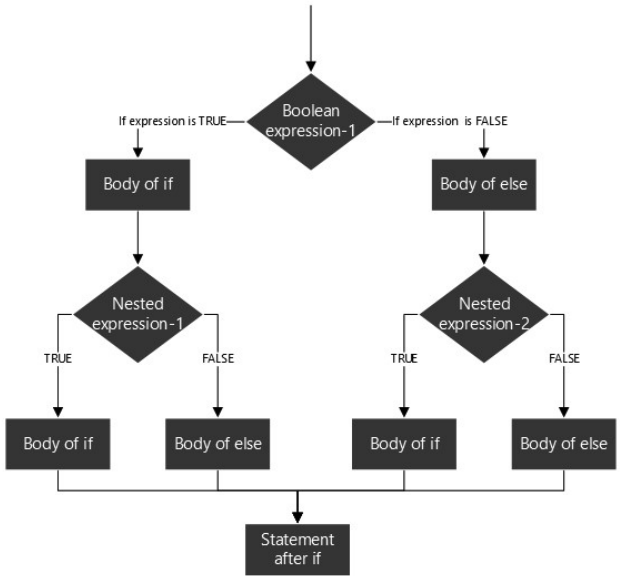
    if(x > 5)
    {
        printf("%d is greater than 5\n",x);
    }
    else if(x > 3)
    {
        printf("%d is greater than 3\n",x);
    }
    else
    {
        printf(" x = %d\n",x);
    }

    return 0;
}

```
#include <stdio.h>

int main(void){

    int x = 9;

    if(x > 5){

        printf("%d is greater than 5\n",x);

    } else if(x > 3){

        printf("%d is greater than 3\n",x);

    } else {

        printf(" x =  %d\n",x);

    }

    return 0;
}
```

```
if (boolean_expression_1)
{
    if(nested_expression_1)
    {
        // If boolean_expression_1 and
        // nested_expression_1 both are true
    }
    else
    {
        // If boolean_expression_1 is true
        // but nested_expression_1 is false
    }
    // If boolean_expression_1 is true
}
else
{
    if(nested_expression_2)
    {
        // If boolean_expression_1 is false
        // but nested_expression_2 is true
    }
    else
    {
        // If both boolean_expression_1 and
        // nested_expression_2 is false
    }

    // If boolean_expression_1 is false
}
```

Nested if...else statement




```
/*
 * C program to find maximum between three numbers
 */

#include <stdio.h>

int main()
{
    /* Declare three integer variables */
    int num1, num2, num3;

    /* Input three numbers from user */
    printf("Enter three numbers: ");
    scanf("%d%d%d", &num1, &num2, &num3);

    if(num1 > num2)
    {
        if(num1 > num3)
        {
            /* If num1 > num2 and num1 > num3 */
            printf("Num1 is max.");
        }
        else
        {
            /* If num1 > num2 but num1 < num3 */
            printf("Num3 is max.");
        }
    }
    else
    {
        if(num2 > num3)
        {
            /* If num1<num2 and num2>num3 */
            printf("Num2 is max.");
        }
        else
        {
            /* If num1 < num2 and num 2< num3 */
            printf("Num3 is max.");
        }
    }

    return 0;
}
```

```
Enter three numbers: 8 9 4
Num2 is max.
```


Building a logical comparison

```
#include <stdio.h>

int main(void)
{
    int coordinate;

    printf("Input target coordinate: ");
    scanf("%d",&coordinate);

    if( coordinate >= -5 && coordinate <= 5 ){
      printf("Close enough!");
    } else {
      printf("Target is out of range!");
    }

    return 0;
}
```

```
Input target coordinate: -1
Close enough!
```

```
Input target coordinate: 7
Target is out of range!
```

```c
/* Checking for a range of values */
#include <stdio.h>

int main(void){
    int x = 0;

    printf("Enter a number from 1 to 10: ");
    scanf("%d", &x);

    if ( x < 1 || x > 10 )
        printf("\nNumber not in range\n");
    else
        printf("\nThank you\n");

    return 0;
}
```

```
Enter a number from 1 to 10: 50

Number not in range
```

```
Enter a number from 1 to 10: 5

Thank you
```


Warning!

- Remember to use parentheses with conditions, otherwise your program may not mean what you think

```c
int   i = 10;

if(!i == 5)
      printf("i is not equal to five\n");
else
      printf("i is equal to five\n");
```


Warning!

- Remember to use parentheses with conditions, otherwise your program may not mean what you think

in this attempt to say “i not equal to five”, “!i” is evaluated first. As “i” is 10, i.e. non zero, i.e. true, “!i” must be false, i.e. zero. Zero is compared with five

```c
int   i = 10;

if(!i == 5)
      printf("i is not equal to five\n");
else
      printf("i is equal to five\n");
```

```
i is equal to five
```

Comparison of a float with a value

```
float x = 0.1;

if (x == 0.1)
    printf("IF");
else
    printf("ELSE");
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Comparison of a float with a value

```
float x = 0.1;

if (x == 0.1)
    printf("IF");
else
    printf("ELSE");
```

ELSE

electronics010.blogspot.com C language Eng. Mohamed Yousef

Comparison of a float with a value

```
float x = 0.1;

if (x == 0.1f)
    printf("IF");
else
    printf("ELSE");
```

OR

```
float x = 0.1;

if (x == (float)0.1)
    printf("IF");
else
    printf("ELSE");
```

Comparison of a float with a value

```
float x = 0.1;

if (x == 0.1f)
    printf("IF");
else
    printf("ELSE");
```

OR

```
float x = 0.1;

if (x == (float)0.1)
    printf("IF");
else
    printf("ELSE");
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Comparison of a float with a value

```
double x = 0.1;

if (x == 0.1)
    printf("IF");
else
    printf("ELSE");
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Comparison of a float with a value

```
double x = 0.1;

if (x == 0.1)
    printf("IF");
else
    printf("ELSE");
```

IF

Comparison of a float with a value

"warning: comparing floating point with == or != is unsafe".

electronics010.blogspot.com C language Eng. Mohamed Yousef

Comparison of a float with a value

The precision of float (4 bytes) is less than the double (8 bytes) .
Hence after promotion of float into double(at the time of comparison) compiler will pad the remaining bits with zeroes.
Hence we get the different result in which decimal equivalent of both would be different.

For instance,

```
In float
=> (0.1)10 = (0.00011001100110011001100)2
In double after promotion of float ...(1)
=> (0.1)10 = (0.00011001100110011001100000000000000000...)2
                                      ^ padding zeroes here
In double without promotion ... (2)
=> (0.1)10 = (0.0001100110011001100110011001100110011001100110011001)2

Hence we can see the result of both equations are different.
Therefore 'if' statement can never be executed.
```

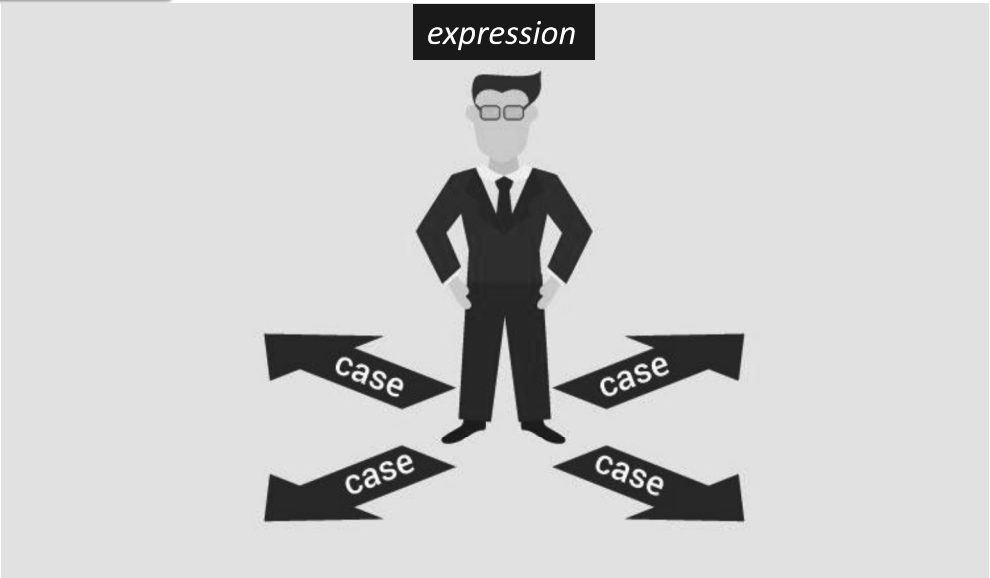
electronics010.blogspot.com C language Eng. Mohamed Yousef

Comparison of a float with a value

There is a difference between 0.1, float(0.1), and double(0.1).
In C/C++ the numbers 0.1 and double(0.1) are the same thing,
but when I say "0.1" in text I mean the exact base-10 number,
whereas float(0.1) and double(0.1) are rounded versions of 0.1.
And, to be clear, float(0.1) and double(0.1) don't have the same value,
because float(0.1) has fewer binary digits, and therefore has more error.
Here are the exact values for 0.1, float(0.1), and double(0.1):

Number	Value
0.1	0.1 (of course)
float(.1)	0.100000001490116119384765625
double(.1)	0.1000000000000000055511151231257827021181583404541015625

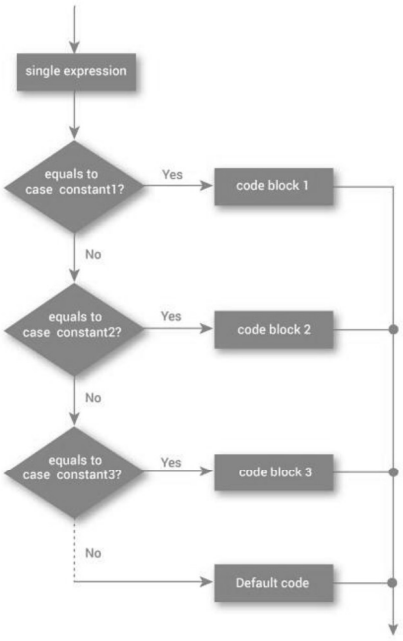
switch...case Statement



electronics010.blogspot.com C language Eng. Mohamed Yousef

switch...case Statement

```
switch(expression)
{
    case value1:
        statement(s);
        break;
    case value2:
        statement(s);
        break;
    case value3:
        statement(s);
        break;
    default:
        statement(s);
}
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

switch...case Statement

```
switch(expression)
{
    case value1:
        statement(s);
        break;
    case value2:
        statement(s);
        break;
    case value3:
        statement(s);
        break;
    default:
        statement(s);
}
```

- The *controlling expression* of a switch statement shall have *integer* type.
- It can be a *variable*, a value returned from a *function*, or a *mathematical* operation.
- The *value* of each case label shall be an *integer constant*.
- The implementation may *limit* the number of case values in a switch statement to *1023 case* labels (C11).

```c
/* Example, the following program uses the switch
   structure to evaluate a user's response
   from a menu.
*/
#include <stdio.h>

int main(void){
    int x = 0;

    printf(" 1) Sports\n");
    printf(" 2) Geography\n");
    printf(" 3) Music\n");
    printf(" Please select a category (1-3): ");
    scanf("%d", &x);

    switch (x) {
        case 1:
            printf("\nYou selected sports\n");
            break;
        case 2:
            printf("You selected geography\n");
            break;
        case 3:
            printf("You selected music\n");
            break;
        default:
            printf("Wrong choice!\n");
    } //end switch

    return 0;
}
```

```
1) Sports
2) Geography
3) Music
Please select a category (1-3): 2
You selected geography
```

```
1) Sports
2) Geography
3) Music
Please select a category (1-3): 5
Wrong choice!
```



```c
#include <stdio.h>

int main(void){
    int x = 0;

    printf(" 1) Sports\n");
    printf(" 2) Geography\n");
    printf(" 3) Music\n");
    printf(" Please select a category (1-3): ");
    scanf("%d", &x);

    switch (x) {
        case 1:  printf("\nYou selected sports\n");  break;
        case 2:  printf("You selected geography\n"); break;
        case 3:  printf("You selected music\n");     break;
        default: printf("Wrong choice!\n");
    } //end switch

    return 0;
}
```



```c
#include <stdio.h>

int main(void){
    int x = 0;

    printf(" Enter number between 1 - 3 : ");
    scanf("%d", &x);

    switch (x) {
        case 1:
        case 2:
            printf("You entered 1 or 2\n");
            break;
        case 3:
            printf("You entered 3\n");
            break;
        default:
            printf("Wrong choice!\n");
    }

    return 0;
}
```

```
Enter number between 1 - 3 : 2
You entered 1 or 2
```

```
Enter number between 1 - 3 : 3
You entered 3
```

```c
#include <stdio.h>

int main(){
int sport = 2;

    switch(sport){
        case 1:
            printf("\n Basketball \n");
            break;
        case 2:
            printf("\n Swimming \n");
            break;
        default:
           printf("\n Else\n");
    }

    return 0;
}
```

```
Swimming
```

```c
#include <stdio.h>

int main(){
int sport = 2;
const int Basketball = 1;
const int Swimming = 2;

    switch(sport){
        case Basketball:
            printf("\n Basketball \n");
            break;
        case Swimming:
            printf("\n Swimming \n");
            break;
        default:
           printf("\n Else\n");
    }

    return 0;
}
```

```
error: case label does not reduce to an integer constant
```



```c
#include <stdio.h>

int main(){
int sport = 2;

    switch(sport){
        case 1:
            printf("\n Basketball \n");
            break;
        case 2:
            printf("\n Swimming \n");
            break;
        default:
           printf("\n Else\n");
    }

    return 0;
}
```

```
Swimming
```

```c
#include <stdio.h>

#define Basketball 1
#define Swimming 2

int main(){
int sport = 2;

    switch(sport){
        case Basketball:
            printf("\n Basketball \n");
            break;
        case Swimming:
            printf("\n Swimming \n");
            break;
        default:
           printf("\n Else\n");
    }

    return 0;
}
```

```
Swimming
```


conditional operator

```c
larger = (a > b) ? a : b;
```

```c
result = comparison ? if_true : if_false;
```

```c
#include <stdio.h>

int main(void){
    int a,b,larger;

    printf("Enter value A: ");
    scanf("%d",&a);

    printf("Enter different value B: ");
    scanf("%d",&b);

    larger = (a > b) ? a : b;
    printf("Value %d is larger.\n",larger);

    return 0;
}
```

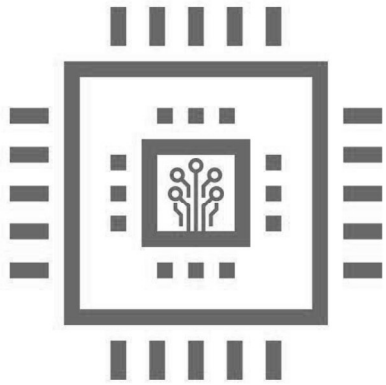
```
Enter value A: 4
Enter different value B: 5
Value 5 is larger.
```

LAB

```c
#include <stdio.h>

int main()
{
    char operator_char;
    double firstNumber,secondNumber;

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator_char);

    printf("Enter two operands: ");
    scanf("%lf %lf",&firstNumber, &secondNumber);

    switch(operator_char)
    {
        case '+':   // ASCII => 43
            printf("%g + %g = %g",firstNumber, secondNumber, firstNumber +
secondNumber);
            break;

        case '-':   // ASCII => 45
            printf("%g - %g = %g",firstNumber, secondNumber, firstNumber -
secondNumber);
            break;

        case '*':   // ASCII => 42
            printf("%g * %g = %g",firstNumber, secondNumber, firstNumber *
secondNumber);
            break;

        case '/':   // ASCII => 47

            if(secondNumber == 0){
                printf("second number is ZERO!\n");
            } else {
                printf("%g / %g = %g",firstNumber, secondNumber,
firstNumber / secondNumber);
            }

            break;

        // operator doesn't match any case constant (+, -, *, /)
        default:
            printf("Error! operator is not correct");
    }

    return 0;
}
```

Embedded Systems

C language



06- Operators - #2

Eng. Mohamed Yousef

Arithmetic Operators

```
/* Post-increment operator */

j = 4;
k = j++;        // k = 4, j = 5

/* Pre-increment operator */

j = 4;
k = ++j;       // k = 5, j = 5

/* Post-decrement operator */

j = 12;
k = j--;     // k = 12, j = 11

/* Pre-decrement operator */

j = 12;
k = --j;    // k = 11, j = 11
```

Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder (integer division)
++	Auto increment
−−	Auto decrement

electronics010.blogspot.com C language Eng. Mohamed Yousef

++ Operator

```
#include <stdio.h>

int main(void){
    int x = 0;
    int y = 0;

    printf("\n The value of x is %d\n", ++x);
    printf(" The value of y is %d\n", y++);


    return 0;
}
```

++ Operator

```c
#include <stdio.h>

int main(void){
    int x = 0;
    int y = 0;

    printf("\n The value of x is %d\n", ++x);
    printf(" The value of y is %d\n", y++);


    return 0;
}
```

```
The value of x is 1
The value of y is 0
```



```c
#include <stdio.h>

int main(void){
    int x = 0;
    int y = 1;

    x = y++ * 2;
    printf("\n The value of x is: %d\n", x);

    x = 0; y = 1;
    x = ++y * 2;
    printf(" The value of x is: %d\n", x);

    return 0;
}
```



```c
#include <stdio.h>

int main(void){
    int x = 0;
    int y = 1;

    x = y++ * 2;
    printf("\n The value of x is: %d\n", x);

    x = 0; y = 1;
    x = ++y * 2;
    printf(" The value of x is: %d\n", x);

    return 0;
}
```

```
The value of x is: 2
The value of x is: 4
```

```c
#include <stdio.h>

int main(void){
    int i = 1;

    printf("\n %d %d %d\n", i++, i++, i);

    return 0;
}
```



```c
#include <stdio.h>

int main(void){
    int i = 1;

    printf("\n %d %d %d\n", i++, i++, i);

    return 0;
}
```

```
2 1 3
```


Even though most, if not all, C compilers will run the preceding code the way you would expect.
Due to ANSI C compliance, the following statement can produce three different results with three different compilers:

```c
anyFunction(++x, x, x++);
```

The argument ++x is NOT guaranteed to be done first before the other arguments (x and x++) are processed.
In other words, *there is no guarantee that each C compiler will process sequential expressions (an expression separated by commas) the same way*.

```c
#include <stdio.h>

int main(void){
    int i = 1;

    printf("\n %d %d %d\n", i++, i++, i);

    return 0;
}
```

```
2 1 3
```

```
2 1 3 - using g++ 4.2.1 on Linux.i686
1 2 3 - using g++ 4.2.1 on SunOS.sun4u
1 2 3 - using SunStudio C++ 5.9 on Linux.i686
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

int main(void){
    int x = 0, y = 0;

    x = y++ * 4;
    printf("\nThe value of x is %d\n", x);

    y = 0;
    x = ++y * 4;
    printf("\nThe value of x is now %d\n", x);

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

int main(void){
    int x = 0, y = 0;

    x = y++ * 4;
    printf("\nThe value of x is %d\n", x);

    y = 0;
    x = ++y * 4;
    printf("\nThe value of x is now %d\n", x);

    return 0;
}
```

```
The value of x is 0

The value of x is now 4
```

-- Operator

```c
#include <stdio.h>

int main(void){
    int x = 1, y = 1;

    x = y-- * 4;
    printf("\n The value of x is %d\n", x);

    y = 1;
    x = --y * 4;
    printf("\n The value of x is now %d\n", x);

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

-- Operator

```c
#include <stdio.h>

int main(void){
    int x = 1, y = 1;

    x = y-- * 4;
    printf("\n The value of x is %d\n", x);

    y = 1;
    x = --y * 4;
    printf("\n The value of x is now %d\n", x);

    return 0;
}
```

```
The value of x is 4

The value of x is now 0
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Assignment Operators

Shorthand assignment operator	Example	Meaning
+=	a += 10	a = a + 10
-=	a -= 10	a = a - 10
*=	a *= 10	a = a * 10
/=	a /= 10	a = a / 10
%=	a %= 10	a = a % 10

Assignment Operators

```c
#include <stdio.h>

int main(void){
    int x = 1, y = 2;

    x += y * x + 1;    // x = x + (y * x + 1);
    printf("\n The value of x is: %d\n", x);

    return 0;
}
```

```
The value of x is: 4
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

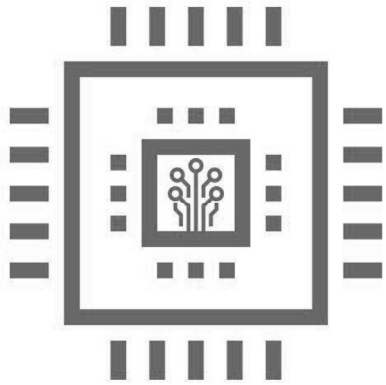
Assignment Operators

```c
#include <stdio.h>

int main(void){
    int x = 1, y = 2;

    x -= y * x + 1;    // x = x - (y * x + 1);
    printf("\n The value of x is: %d\n", x);

    return 0;
}
```

```
The value of x is: -2
```

Embedded Systems

C language



07- Loop

Eng. Mohamed Yousef

Loops

You may encounter situations, when a block of code needs to be executed several number of times.
A loop statement allows us to execute a statement or group of statements multiple times.

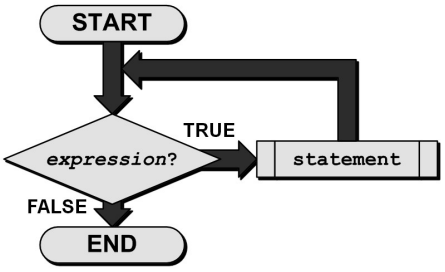
There are three loops in C programming:

- for loop
- while loop
- do...while loop

electronics010.blogspot.com C language Eng. Mohamed Yousef

while loop

```
while(condition)
{
    statement(s);
}
```



A *while* loop can also forgo the curly brackets when it has only one statement

```
while(condition)
    statement;
```

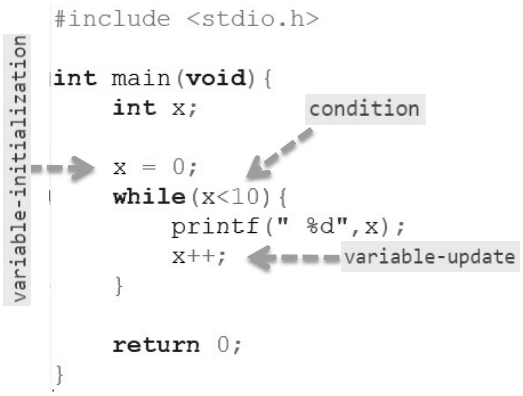
while loop

```c
#include <stdio.h>

int main(void){
    int x;

    x = 0;
    while(x<10){
        printf(" %d",x);
        x++;
    }

    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

while loop

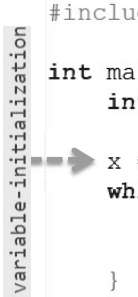



```c
#include <stdio.h>

int main(void){
    int x;

    x = 0;
    while(x<10){
        printf(" %d",x);
        x++;
    }

    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

while loop




```c
#include <stdio.h>

int main(void){
    int x;

    x = 11;
    while(x<10){
        printf(" %d",x);
        x++;
    }

    return 0;
}
```

while loop

```
#include <stdio.h>

int main(void){
    int x;

    x = 0;
    while(x<10){
        printf(" %d",x);
    }

    return 0;
}
```

variable-update

```
0 0 0 0 0 0 0 0 0 0 0 0
```

INFINITE LOOPS

Infinite loops **are loops that never end. They are created when a loop's expression is never set to exit the loop.**

To exit an infinite loop, press *Ctrl+C*, which produces a break in the program.
If this does not work, you may need to end the task.

electronics010.blogspot.com C language Eng. Mohamed Yousef

while loop

Looping endlessly but on purpose

Occasionally, a program needs an endless loop.

For example, a *microcontroller* may load a program that runs as long as the device is on.

```
while(1)
```

The value in the parentheses *doesn't necessarily need to be 1*; any *True* or *non-zero* value works.

When the loop is endless on purpose, most programmers set the value to 1 simply to self-document that they know what's up.

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
#include <stdio.h>

int main(void){
    int x;

    x = 0;
    while(x<10)
        printf("\n The value of x is %d", x++);

    return 0;
}
```

```
The value of x is 0
The value of x is 1
The value of x is 2
The value of x is 3
The value of x is 4
The value of x is 5
The value of x is 6
The value of x is 7
The value of x is 8
The value of x is 9
```

Example

Calculate the Summation of odd values between 1 and 99

```
#include <stdio.h>

int main(void){
    int i = 1, sum = 0;     // Initiation is placed here

    while(i <= 99){
        sum += i;
        i+=2;               // Increment is placed here
    }

    printf(" Summation of odd values between (1 and 99) is : %d", sum);

    return 0;
}
```

```
Summation of odd values between (1 and 99) is : 2500
```


Example

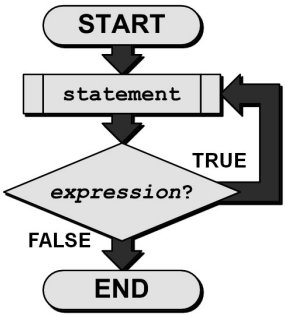
Calculate the Average Students Degrees

```
#include <stdio.h>

int main(void){
    int students_number = 0;
    float degree = 1.0, sum = 0.0;

    printf("\n Enter negative value to exit:\n");

    while(degree > 0){

        students_number++;
        printf(" Enter student (%d) degree : ", students_number);
        scanf("%f", &degree);

        sum += degree;

    }

    sum -= degree;
    students number--;
    if(students_number != 0){
        printf("\n Average student degree is : %g", sum / students_number);
    }
    return 0;
}
```

```
Enter negative value to exit:
Enter student (1) degree : 25
Enter student (2) degree : 34
Enter student (3) degree : 18
Enter student (4) degree : 49
Enter student (5) degree : -1
Average students degree is : 31.5
```


do .. while loop

```
do
{
    statement(s);
} while (condition);
```



The *do..while* loop is similar to the *while* loop with one important difference.
The body of *do...while* loop is *executed once, before checking the test expression*.
Hence, the *do...while* loop is *executed at least once*.

In the do while loop's last statement, the ending *brace* comes before the while statement, and the while statement must end with a *semicolon*.

Example Calculate Polynomial Value

Following program evaluates the polynomial $f(x) = 5X^2 + 3X + 2$

```c
#include <stdio.h>

int main(void){
    float x = 0.0, y = 0.0;
    char ans = '\0';


    do{
        printf("\n Enter x value : ");
        scanf("%f", &x);

        y = 5*x*x + 3*x + 2;

        printf(" y(%g) = %g", x, y);

        printf("\n Do you want to evaluate again (y/n)? ");
        scanf(" %c", &ans);    // space before format specifier.

    } while(ans == 'y');

    return 0;
}
```

```
Enter x value : 3
y(3) = 56
Do you want to evaluate again (y/n)? y

Enter x value : 5
y(5) = 142
Do you want to evaluate again (y/n)? n

Process returned 0 (0x0)   execution time : 19.678 s
Press any key to continue.
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example Calculate Polynomial Value

Following program evaluates the polynomial $f(x) = 5X^2 + 3X + 2$

```c
#include <stdio.h>

int main(void){
    float x = 0.0, y = 0.0;
    char ans = '\0';

    do{
       // printf("\n Enter x value : ");
       // scanf("%f", &x);

        y = 5*x*x + 3*x + 2;
        printf(" y(%g) = %g", x, y);

        printf("\n Do you want evaluate again (y/n)? ");
        scanf("%c", &ans);


    } while(ans == 'y');


    return 0;
}
```

```
y(0) = 2
Do you want evaluate again (y/n)? y
y(0) = 2
Do you want evaluate again (y/n)?
Process returned 0 (0x0)   execution time : 6.949 s
Press any key to continue.
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

When you enter an 'n' or 'y' (ie a single character) and then press "Enter" key , there are actually two characters sitting in the input buffer: the single-character you typed and a newline character \n ("Enter").
If y is the input character, then input buffer will contain y\n.

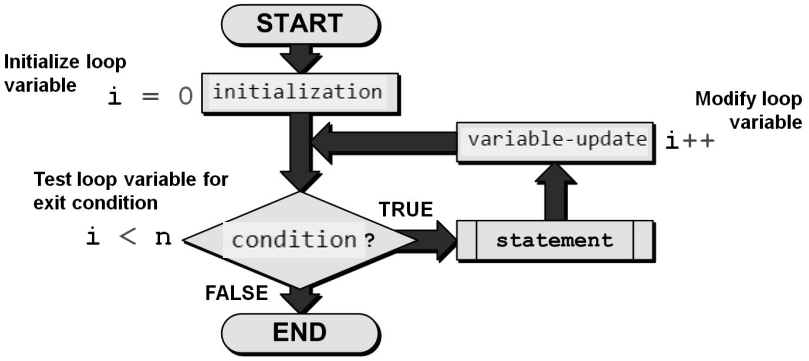
First scanf will read y leaving \n in the buffer for next call of scanf.
On entering the loop, your second scanf will read this leftover \n from input buffer and loop runs one more time without waiting for your input

A space before %c will eat up this \n left behind by scanf

for loop

```
for (initializationStatement; testExpression; updateStatement)
{
        // codes
}
```

```
for(i = 0; i < 10; i++){
    // statements
}
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

for loop

```
#include <stdio.h>

int main(void){
    int counter = 0;

    for(counter = 1; counter <= 5; counter++){

        printf("\n counter = %d", counter);
    }

    return 0;
}
```

```
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

for loop

Increment Expression Acts Like a Standalone Statement

The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`. Therefore, the expressions are all equivalent in the increment part of the `for` statement.

```
counter = counter + 1
counter += 1
++counter
counter++
```

for loop

```c
#include <stdio.h>

int main(void){
    float counter = 0;

    for(counter = 1; counter <= 5; counter += 0.5){

        printf("\n counter = %g", counter);
    }

    return 0;
}
```

```
counter = 1
counter = 1.5
counter = 2
counter = 2.5
counter = 3
counter = 3.5
counter = 4
counter = 4.5
counter = 5
```


for loop

```c
#include <stdio.h>

int main(void){
    int counter = 0;

    for(counter = 5; counter >= 1; counter--){

        printf("\n counter = %d", counter);
    }

    return 0;
}
```

```
counter = 5
counter = 4
counter = 3
counter = 2
counter = 1
```


Extending the for Loop

```c
#include <stdio.h>

int main(void){
    int counter, j, k;

    for(counter = 1, j = 5, k = -1; counter <= 5; counter++, j++, k--){

        printf("\n counter = %d, j = %d, k = %d", counter, j, k);
    }

    return 0;
}
```

```
counter = 1, j = 5, k = -1
counter = 2, j = 6, k = -2
counter = 3, j = 7, k = -3
counter = 4, j = 8, k = -4
counter = 5, j = 9, k = -5
```

Extending the for Loop

```c
#include <stdio.h>

int main(void){
    int counter = 1;

    for(; counter <= 5; counter++){

        printf("\n counter = %d", counter);
    }

    return 0;
}
```

```
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Extending the for Loop

```c
#include <stdio.h>

int main(void){
    int counter = 1;

    for(; counter <= 5;){

        printf("\n counter = %d", counter);
        counter++;
    }

    return 0;
}
```

```
counter = 1
counter = 2
counter = 3
counter = 4
counter = 5
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Extending the for Loop

```c
#include <stdio.h>

int main(void){

    for(;;){

        printf("\n Endless Loop!");

    }

    return 0;
}
```

```
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
Endless Loop!
```

Example

Calculate the Average Students Degrees

```
#include <stdio.h>

int main(void){
    int students_number = 0, counter = 0;
    float degree = 0.0, sum = 0.0;

    printf("\n Enter the number of the students:");
    scanf("%d", &students_number);

    for(counter = 1; counter <= students_number; counter++){

        printf(" Enter student (%d) degree:", counter);
        scanf("%f", &degree);
        sum += degree;
    }

    printf(" Average students degree is : %g\n", sum / students_number);

    return 0;
}
```

```
Enter the number of the students:4
Enter student (1) degree:45.5
Enter student (2) degree:37
Enter student (3) degree:28
Enter student (4) degree:43
Average students degree is : 38.375
```


Nested loops

Nested while loop

```
while(condition)
{
    // statements

    while(condition)
    {
        // Inner loop statements
    }

    // statements
}
```

Nested do .. while loop

```
do
{
    // statements

    do
    {
        // Inner loop statements
    }while(condition);

    // statements
}while(condition);
```


Nested loops

Nested for loop

```
for(initialization; condition; update)
{
    // statements

    for(initialization; condition; update)
    {
        // Inner loop statements
    }

    // statements
}
```

Example

```c
#include <stdio.h>

int main(void){
    int outer = 0, inner = 0;

    for(outer = 1; outer <= 10; outer++){

        for(inner = 1; inner <= 10; inner++){

            printf(" %d * %d = %d \n", outer, inner, outer*inner);
        }

        printf("--------------------\n");
    }



    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
--------------------
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
--------------------
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
--------------------
```

```
4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20
4 * 6 = 24
4 * 7 = 28
4 * 8 = 32
4 * 9 = 36
4 * 10 = 40
--------------------
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
--------------------
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
6 * 10 = 60
--------------------
```

```
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
--------------------
8 * 1 = 8
8 * 2 = 16
8 * 3 = 24
8 * 4 = 32
8 * 5 = 40
8 * 6 = 48
8 * 7 = 56
8 * 8 = 64
8 * 9 = 72
8 * 10 = 80
--------------------
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
9 * 10 = 90
--------------------
```

```
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
--------------------
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

for loop

```c
#include <stdio.h>

int main(void){
    int x;

    for(x = 0; x = 10; x++){

        printf(" x = %d\n", x);
    }

    return 0;
}
```

```
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
x = 10
```

for loop

```c
#include <stdio.h>

int main(void){
    int x;

    for(x = 0; x == 10; x++){

        printf(" x = %d\n", x);
    }

    return 0;
}
```

```
Process returned 0 (0x0)   execution time : 0.125 s
Press any key to continue.
```


goto

goto is a jump statement used to transfer program control unconditionally from one part of a function to another.

You can transfer program control from one position to any position within a function.

goto

Syntax of *goto* statement

```c
    goto lable1;
    statements;

lable1:
    statements;
```

```c
lable1:
    statements;

    goto lable1;
    statements;
```

goto

```
#include <stdio.h>

int main(void){
    char gender = '\0';

    printf("\n Enter your gender (f/m) : ");
    scanf("%c", &gender);

    if(gender == 'f'){
        goto female;
    } else if(gender == 'm'){
        goto male;
    } else{
        goto other;
    }

female:
    printf("\n Good morning Miss.");
    goto finsh;

male:
    printf("\n Good morning Mr. ");
    goto finsh;

other:
printf("\n Thanks ");

finsh:
    return 0;
}
```

```
Enter your gender (f/m) : f

Good morning Miss.
```

```
Enter your gender (f/m) : m

Good morning Mr.
```

```
Enter your gender (f/m) : k

Thanks
```


break Statement

The *break* statement terminates the loop (for, *while and do...while loop*) immediately when it is encountered.

The break statement is used with decision making statement such as *if...else*.

```
break;
```

```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```


break Statement

```
#include <stdio.h>

int main(void){
    int x = 0;

    for(x=1; x <= 10; x++){

        printf("%d ", x);

        if(x == 5) break;
    }

    printf("\n Thanks");

    return 0;
}
```

```
1 2 3 4 5
 Thanks
```

continue Statement

The *continue* statement skips some statements inside the loop.

The *continue* statement is used with decision making statement such as *if...else*.

```
continue;
```

```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

continue Statement

```c
#include <stdio.h>

int main(void){
    int x = 0;

    for(x=1; x <= 10; x++){

       if(x == 5) continue;

       printf("%d ", x);
    }

    printf("\n Thanks");

    return 0;
}
```

```
1 2 3 4 6 7 8 9 10
 Thanks
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

Example **Calculate the Average Students Degrees**

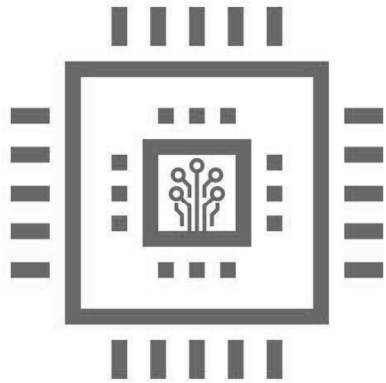
```c
#include <stdio.h>
#define true    1

int main(void){
    int students_number = 0;
    float degree = 0.0, sum = 0.0;

    printf("\n Enter negative number to finish. \n");

    while(true){

        students_number++;

        printf(" Enter student %d degree : ", students_number);
        scanf("%f", &degree);

        if(degree < 0) break;

        sum += degree;
    }

    students_number--;

    if(students_number != 0){
        printf("\n Average student degree is : %g", sum / students_number);
    }
    return 0;
}
```

```
Enter negative number to finish.
Enter student 1 degree : 8
Enter student 2 degree : 7
Enter student 3 degree : 9
Enter student 4 degree : 6
Enter student 5 degree : -1
```

```
/*
================================================================================
==
PROGRAM     :
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/

#include <stdio.h>
#define true    1

int main(void){
    int students_number = 0;
    float degree = 0.0, sum = 0.0;

    printf("\n Enter negative number to finish. \n");

    while(true){

        students_number++;

        printf(" Enter student %d degree : ", students_number);
        scanf("%f", &degree);

        if(degree < 0) break;

        sum += degree;
    }

    students_number--;

    if(students_number != 0){
        printf("\n Average student degree is : %g", sum / students_number);
    }
    return 0;
}
```

Embedded Systems

C language

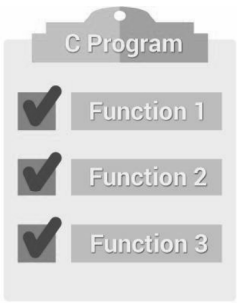


08- Functions

Eng. Mohamed Yousef

Introducion

- **Programmers seldom write programs as one long series of steps**
- **Instead, they break the programming problem down into reasonable units and tackle one small task at a time**
- **These reasonable units are called functions**
- **Programmers also refer to them as subroutines, or procedures**
- **The name that programmers use for their units usually reflects the programming language they use**



Functions
Self contained program segments designed to perform a specific, well defined task.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Introducion

Advantages of dividing program into functions

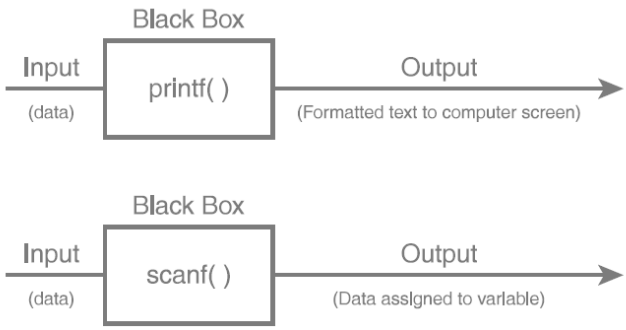
- A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.
- **Reusability** of code. Functions once defined can be used any several times. You can use functions of one program in another program. It saves time and effort.
- Code **maintenance and debugging** is easier. In case of errors in a function, you only need to debug that particular function instead of debugging entire program.

Introducion

Advantages of dividing program into functions

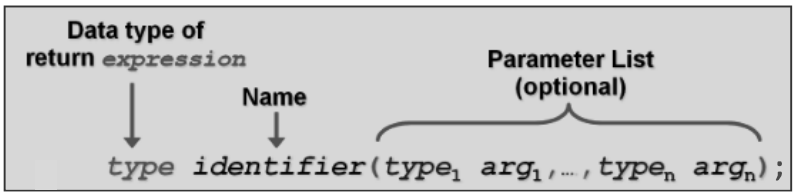
- **Information Hiding**

  Functions can be seen as black boxes. You don't know how the black box performs (implements) the task; you just simply know it works when needed.



Function Prototypes (function declaration)

A *function prototypes (declaration)* tells the compiler about a function's name, return type, and parameters.



Function Prototypes (function declaration)

```
float addTowNumbers (float, float);
```

This function prototype tells C the following things about the function:

- The data type returned by the function—in this case a `float` data type is returned
- The number of parameters received—in this case two
- The data types of the parameters—in this case both parameters are float data types
- The order of the parameters

Function Prototypes (function declaration)

Function implementations and their prototypes can vary. It is not always necessary to send input as parameters to functions, nor is it always necessary to have functions return values. In these cases, programmers say the functions are void of parameters and/or are void of a return value. The next two function prototypes demonstrate the concept of functions with the `void` keyword.

```
void printBalance(int);  //function prototype

int createRandomNumber(void);  //function prototype
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Function Prototypes (function declaration)

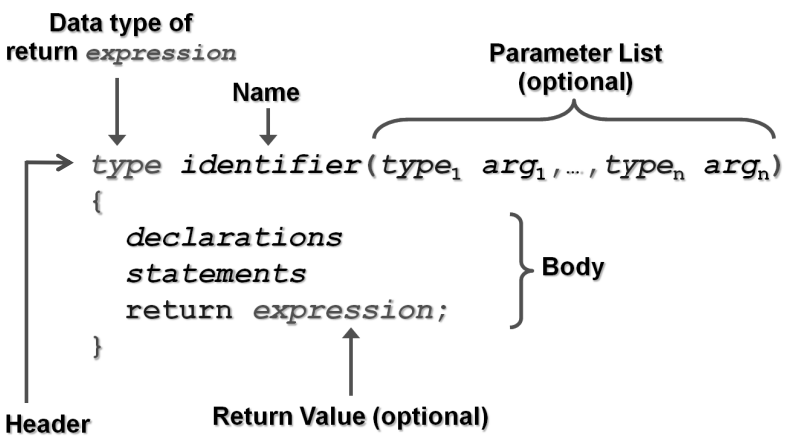
Function prototypes should be placed outside the `main()` function and before the `main()` function starts, as demonstrated next.

```
#include <stdio.h>

int addTwoNumbers(int, int);  //function prototype

main()
{

}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

FUNCTION DEFINITIONS

A *function definition* provides the *actual body* of the function.



FUNCTION DEFINITIONS

```
#include <stdio.h>

int addTwoNumbers(int, int);  //function prototype

main()

{

   printf("Nothing happening in here.");

}

//function definition
int addTwoNumbers(int operand1, int operand2)
{
   return operand1 + operand2;

}
```


Return Statement

The return statement:

- terminates the execution of a function
- returns a value to the calling function

Syntax of return statement

```
return (expression);
```

For example,

```
return a;
return (a+b);
```


Return Statement

```c
int maximum(int x, int y)
{
  int z;

  z = (x >= y) ? x : y;
  return z;
}
```

```c
int bigger(int a, int b)
{
    if (a > b)
        return 1;
    else
        return 0;
}
```

```c
int maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```

This one is more efficient since it doesn't require the local variable `z`.

Return Statement

The function *type* is `void` if

- The `return` statement has no *expression*
- The `return` statement is not present at all

```c
void identifier(type1 arg1,...,typen argn)
{
  declarations
  statements
  return;
}
```

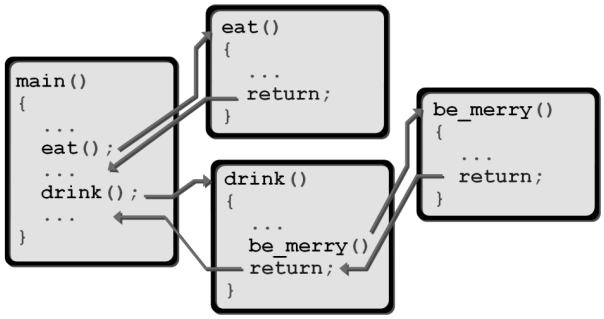

FUNCTION CALLS

➢ To use a function, you will have to call that function to perform the defined task.

➢ When a program calls a function, the program control is transferred to the called function.

➢ A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

➢ To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.



FUNCTION CALLS

```c
#include <stdio.h>

int addTwoNumbers(int, int); //function prototype

int main(){
    int result = 0;

    result = addTwoNumbers(5, 6);

    printf("%d", result);
    return 0;
}

//function definition
int addTwoNumbers(int operand1, int operand2){
    return operand1 + operand2;
}
```

FUNCTION CALLS

```c
#include <stdio.h>

int addTwoNumbers(int, int); //function prototype

int main(){
    printf("%d", addTwoNumbers(5, 6));
    return 0;
}

//function definition
int addTwoNumbers(int operand1, int operand2){
    return operand1 + operand2;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

FUNCTION CALLS

```c
#include <stdio.h>

int addTwoNumbers(int, int); //function prototype

int main(){
    int num1 = 4, num2 = 3;
    printf("%d", addTwoNumbers(num1, num2));
    return 0;
}

//function definition
int addTwoNumbers(int operand1, int operand2){
    return operand1 + operand2;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

FUNCTION CALLS

```c
#include <stdio.h>

void messages(void); //function prototype

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

//function definition
void messages(void){
    printf("Hello World.");
}
```

Function Arguments

- 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.
- While calling a function, there are two ways in which arguments can be passed to a function:
  - *Call by value*
  - *Call by reference*

```
// Formal parameters
void add(int num1, int num2) // Function definition
{
    // Function body
}


int main()
{
        // Actual parameters
    add(10, 20); // Function call

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Function Arguments

Call by value

- Call by value is the default mechanism to pass arguments to a function.
- In Call by value, during function call actual parameter value is copied and passed to formal parameter.
- Changes made to the formal parameters does not affect the actual parameter.

```
int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Function Arguments

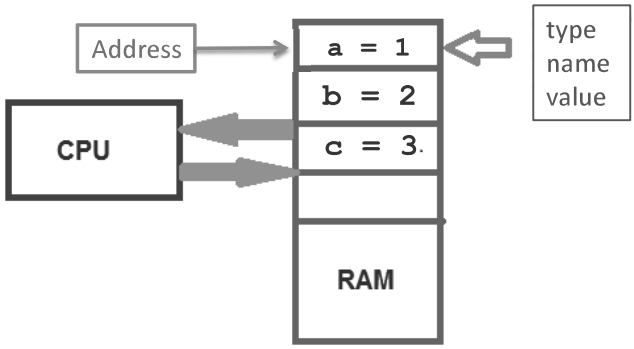
```c
#include <stdio.h>

void By_Value(int x); //function prototype

int main(){
    int a = 6;

    printf("\n a before calling = %d", a);
    By_Value(a);
    printf("\n a after calling = %d", a);

    return 0;
}

//function definition
void By_Value(int x){
    x++;
}
```

```
a before calling = 6
a after calling = 6
```

Function Arguments

Call by reference

- In Call by reference we pass memory location (reference) of actual parameter to formal parameter.
- It uses pointers to pass reference of an actual parameter to formal parameter.
- Changes made to the formal parameter immediately reflects to actual parameter.



Function Arguments

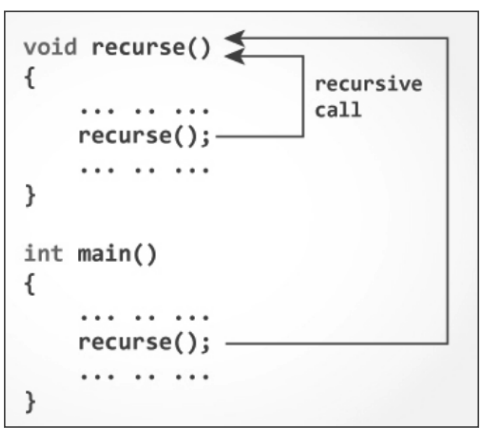
```c
#include <stdio.h>

void  By_Ref (int *x);

int main(){
    int a = 6;

    printf("\n a before calling = %d", a);
    /*
        &a indicates pointer to a ie. address of variable a
    */
    By_Ref(&a);
    printf("\n a after calling = %d", a);

    return 0;
}

//function definition
void  By_Ref (int *x){

    *x = 10;
}
```

```
a before calling = 6
a after calling = 10
```


Recursion

Until now, we called a function from another function.

However, C language allows a function to call itself known as *Recursive function*.



Recursion

A function must not go in indefinite recursion.

Every recursive function must have a base condition to terminate (if...else statement).

electronics010.blogspot.com C language Eng. Mohamed Yousef

Recursion

```c
#include <stdio.h>

void Message(int down_counter);

int main(void){
    Message(5);
    return 0;
}

void Message(int down_counter){

    printf(" Hello\n");
    down_counter--;

    if(down_counter>0){
        Message(down_counter);
    }
}
```

```
Hello
Hello
Hello
Hello
Hello
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Recursion

Advantages of recursion

Using recursion many complex mathematical problems can be solved easily.

Disadvantages of recursion

- Due to incremental functional call, it consumes more memory and takes more time than its iterative approach.
- Recursive programs may crash due to stack overflow (memory shortage) errors.
- Recursive functions are complex to read, write and understand.

Not recommended for Embedded System due to the microcontroller's stack and memory limitations.

Example

Find factorial of a number using recursion

Factorial

From Wikipedia, the free encyclopedia

In mathematics, the **factorial** of a non-negative integer *n*, denoted by *n*!, is the product of all positive integers less than or equal to *n*. For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

The value of 0! is 1, according to the convention for an empty product.[1]

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

Find factorial of a number using recursion

```c
#include <stdio.h>

/* Function declaration */
unsigned long long fact(unsigned int num);

int main(void){
    unsigned int num;
    unsigned long long factorial;

    /* Input an integer from user */
    printf(" Enter any number: ");
    scanf("%u", &num);

    factorial = fact(num);  // Call factorial function

    printf(" Factorial of %u is %llu", num, factorial);

    return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

Find factorial of a number using recursion

```c
/*
  Function to compute and return factorial of any number
  recursively.
 */
unsigned long long fact(unsigned int num){
    // Base condition
    if(num == 0){
        return 1;
    } else {
        return (num * fact(num - 1));
    }
}
```

```
Enter any number: 5
Factorial of 5 is 120
```

```
fact 5 is calculated as follows:
fact (5) = 5 * fact (4) i.e. there is call to fact (4) \\ A
fact (4) = 4 * fact (3)
fact (3) = 3 * fact (2)
fact (2) = 2 * fact (1)
fact (1) = 1 * fact (0)
fact (0) = 1
```

Creating and Using Macros

```
#include <stdio.h>

int Area(int,int);

int main(){
int length = 0, width = 0;

    printf(" length: ");
    scanf("%d", &length);

    printf(" width: ");
    scanf("%d", &width);

    printf(" Area = %d", Area(length,width));

    return 0;
}

int Area(int length,int width){
    return (length*width);
}
```

```
length: 6
width: 4
Area = 24
```

```
#include <stdio.h>

#define AREA(length,width) (length*width)

int main(){
int length = 0, width = 0;

    printf(" length: ");
    scanf("%d", &length);

    printf(" width: ");
    scanf("%d", &width);

    printf(" Area = %d", AREA(length,width));

    return 0;
}
```

```
length: 6
width: 4
Area = 24
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Creating and Using Macros

#include <stdio.h>

int Area(int,int);

int main(){
int length = 0, width = 0;

    printf(" length: ");
    scanf("%d", &length);

    printf(" width: ");
    scanf("%d", &width);

    printf(" Area = %d", Area(length,width));

    return 0;
}

int Area(int length,int width){
    return (length*width);
}

length: 6
width: 4
Area = 24

#include <stdio.h>

#define AREA(length,width) (length*width)

int main(){
int length = 0, width = 0;

    printf(" length: ");
    scanf("%d", &length);

    printf(" width: ");
    scanf("%d", &width);

    printf(" Area = %d", AREA(length,width));

    return 0;
}

length: 6
width: 4
Area = 24

NOTE

Because a function macro looks similar to a function call it can be difficult to tell macro functions and regular functions apart. It is good coding practice to use upper case for all macro names so they are easily distinguished from functions code.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Creating and Using Macros

Macro Expansion

There is a difference between passing expressions to macros and passing them to functions.

When you pass expressions to functions they are first evaluated and the resulting values are received by the function.

As the preprocessor simply performs text replacement; it does not evaluate expressions passed to a macro.

For this reason you must use macros carefully.

For example, here is a common macro error:

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

Consider the following call to SQUARE:

```
someInt = SQUARE(a+1);   // before expansion
someInt = a+1 * a+1;     // after expansion
```

C precedence rules produce an unintended result from this calculation.

The use of parentheses is important in a macro definition using expressions.

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

Creating and Using Macros

Macro Expansion

Even with parentheses, using SQUARE as follows will produce unexpected results:

```
#include <stdio.h>

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

int Square (int x){
    return (x*x);
}

int main(){
int a = 5;
    printf("\n SQUARE = %d", SQUARE(a++));

    a = 5;
    printf("\n SQUARE = %d", ((a++) * (a++)));

    a = 5;
    printf("\n Square = %d", Square(a++));

    return 0;
}
```

```
SQUARE = 30
SQUARE = 30
Square = 25
```

Because a is not evaluated in the same manner as it would be in a function call, it is evaluated twice at compile time and a is incremented before the multiplication.

If SQUARE were a function, a would have been evaluated once at compile time and the resulting value passed to the function.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Creating and Using Macros

White Space in the Preprocessor

Unlike the C compiler, white space is very important to the preprocessor.

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

Second line is defined as an symbolic constant, not as a function macro like first line as intended.

electronics010.blogspot.com C language Eng. Mohamed Yousef

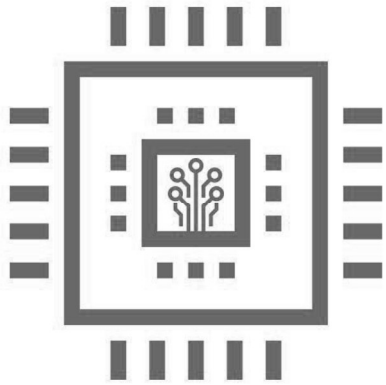
LAB

Write a program to calculate the circle area by using function

```
/*
================================================================================
==
PROGRAM     : circle Area
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/
#include <stdio.h>

float Circle_Area(float radius);

/* ----------------------------------------------------------------------------
*/

int main(){
    float radius = 0.0, area = 0.0;

    printf("\n Enter the radius of Circle : ");
    scanf("%f", &radius);

    area = Circle_Area(radius);
    printf(" Area of Circle : %f\n", area);

    return 0;
}

/* ----------------------------------------------------------------------------
*/

float Circle_Area(float radius){
    return (radius * radius * 3.1416);
}

/* ----------------------------------------------------------------------------
*/
```

Embedded Systems

C language



09- Scope Rules

Eng. Mohamed Yousef

scope

A scope is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.

There are two places where variables can be declared in C programming language:

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Local Variables

Variables that are declared inside a function or block are called local variables.
Block is a sequence of statements grouped together inside a pair of curly braces **{**and **}**.

Properties of a local variable:

- local variable is allocated on C stack.
- Local variables are uninitialized by default and contains garbage value.
- Local variables are not known to functions outside their own.
- They can be used only by statements that are inside that function or block of code.
- Local variables declared inside a block can be accessed only within same or inner blocks.
- Data stored in the variable is lost when the function is terminated (***lifetime***).

Local Variables

```c
#include <stdio.h>

void func(void);

int main(void){

    printf("%d", x);

    return 0;
}


void func(void){
    int x = 10;
}
```

File	Line	Message
C:\Users\Eng...		In function 'main':
C:\Users\Eng...	9	error: 'x' undeclared (first use in this function)

Local Variables

```c
#include <stdio.h>

void func(void);

int main(void){
    func();
    return 0;
}

void func(void){
    int x = 10;

    printf("\n Before block %d", x);

    {
        printf("\n From block %d", x);
    }
}
```

```
Before block 10
From block 10
```


Local Variables

```c
#include <stdio.h>

void func(void);

int main(void){
    func();
    return 0;
}

void func(void){


    printf("\n Before block %d", x);

    {
        int x = 10;
        printf("\n From block %d", x);
    }
}
```

```
In function 'func':
error: 'x' undeclared (first use in this function)
```

Local Variables

```c
#include <stdio.h>

void func(void);

int main(void){
    func();
    return 0;
}

void func(void){

    {
        int x = 10;
        printf("\n From block %d", x);
    }

    printf("\n After block %d", x);
}
```

```
In function 'func':
error: 'x' undeclared (first use in this function)
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Local Variables

```c
#include <stdio.h>

void func(void);

int main(void){
    func();
    return 0;
}

void func(void){

    {
        int x = 10;
        printf("\n From block 1 %d", x);
    }

    {
        printf("\n From block 2 %d", x);
    }
}
```

```
In function 'func':
error: 'x' undeclared (first use in this function)
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Local Variables

```c
#include <stdio.h>

void func(void);

int main(void){
    func();
    return 0;
}

void func(void){

    {
        int x = 10;
        printf("\n From block 1 %d", x);
    }

    {
        int x = 5;
        printf("\n From block 2 %d", x);
    }
}
```

```
From block 1 10
From block 2 5
```

Local Variables

```c
#include <stdio.h>
void func(void);

int main(void){
    func();
    return 0;
}

void func(void){
    {
        int x = 10;
        printf("\n From block 1 %d", x);
        {
            printf("\n From block 2 %d", x);
        }

    }

}
```

```
From block 1 10
From block 2 10
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Local Variables

```c
#include <stdio.h>
void func(void);

int main(void){
    func();
    return 0;
}

void func(void){
int x = 15;
    {
        int x = 10;
        printf("\n From block 1 %d", x);
    }

    {
        int x = 5;
        printf("\n From block 2 %d", x);
    }
    printf("\n Out of blocks %d", x);
}
```

```
From block 1 10
From block 2 5
Out of blocks 15
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Local Variables

```c
#include <stdio.h>
void func(void);

int main(void){
    func();
    return 0;
}

void func(void){
int x = 15;
    {
        int x = 10;
        printf("\n From block 1 %d", x);
    }

    {
        printf("\n From block 2 %d", x);
    }

    printf("\n Out of blocks %d", x);
}
```

```
From block 1 10
From block 2 15
Out of blocks 15
```

Local Variables

```
#include <stdio.h>
void func1(void);
void func2(void);

int main(void){
    func1();
    func2();
    return 0;
}

void func1(void){
int x = 15;
    printf("\n func1 %d", x);
}

void func2(void){
int x = 10;
    printf("\n func2 %d", x);
}
```

```
func1 15
func2 10
```


Global Variables

Global variables are variables declared outside a function, usually on top of the program..

Properties of a global variable:

- Global variable is allocated on data segment .
- Global variable is accessible to all functions of the program.
- Global variables are best suited when most of your functions share common variables.
- Global variables hold their values throughout the ***lifetime*** of your program.
- Global variables are initialized automatically by the system

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

Global Variables

```
#include <stdio.h>
int x = 5;
void func(void);

int main(void){
    printf("\n From main() : %d", x);
    func();

    return 0;
}

void func(void){
    printf("\n From func() : %d", x);
}
```

```
From main() : 5
From func() : 5
```

Global Variables

```c
#include <stdio.h>
int x = 5;
void func(void);

int main(void){
    printf("\n From main() : %d", x);
    func();

    return 0;
}

void func(void){
int x = 10;
    printf("\n From func() : %d", x);
}
```

```
From main() : 5
From func() : 10
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Global Variables

```c
#include <stdio.h>
int x = 5;
void func(void);

int main(void){
    printf("\n Before function call : %d", x);
    func();
    printf("\n After function call : %d", x);

    return 0;
}

void func(void){
    x = 10;
}
```

```
Before function call : 5
After function call : 10
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

```c
#include <stdio.h>
int x = 5;    // global variable
void func(int);

int main(void){
    printf("\n Before function call : %d", x);

    int x = 10; // local variable declaration
    func(x);
    printf("\n After function call : %d", x);

    return 0;
}

void func(int x){
    printf("\n From function : %d", x);
    x = 15;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

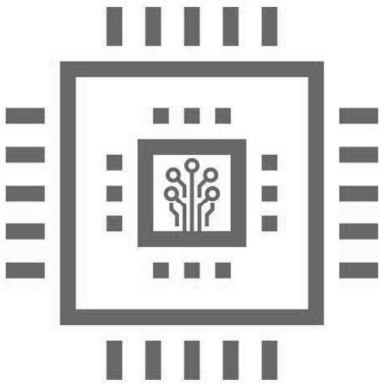
```c
#include <stdio.h>
int x = 5;    // global variable
void func(int);

int main(void){
    printf("\n Before function call : %d", x);

    int x = 10; // local variable declaration
    func(x);
    printf("\n After function call : %d", x);

    return 0;
}

void func(int x){
    printf("\n From function : %d", x);
    x = 15;
}
```

```
Before function call : 5
From function : 10
After function call : 10
```

```c
#include <stdio.h>

int x = 5;    // global variable

void func(int);

int main(void){
    printf("\n Before function call : %d", x);

    int x = 10; // local variable declaration
    func(x);
    printf("\n After function call : %d", x);

    return 0;
}

void func(int x){
    printf("\n From function : %d", x);
    x = 15;
}
```

Embedded Systems

C language



10- Memory Layout

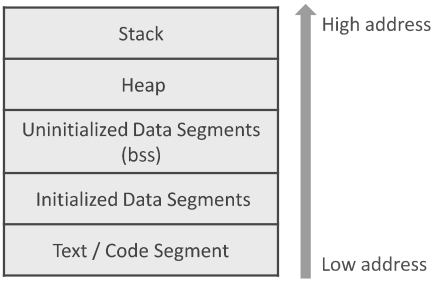
Eng. Mohamed Yousef

Memory Layout of C Programs

When you run any C-program, its executable image is loaded into RAM of computer according to the availability in an organized manner by an Operating System component called Program Loader.

This memory layout is organized in following fashion:

1) *Text or Code Segment*
2) *Initialized Data Segments*
3) *Uninitialized Data Segments (bss)*
4) *Heap Segment*
5) *Stack Segment*

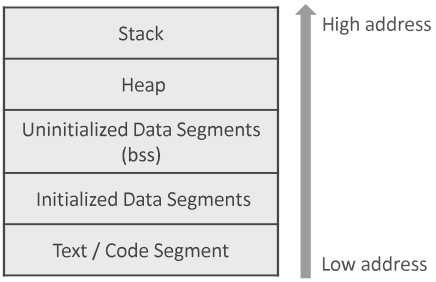


electronics010.blogspot.com C language Eng. Mohamed Yousef

Memory Layout of C Programs

1) Text or Code Segment

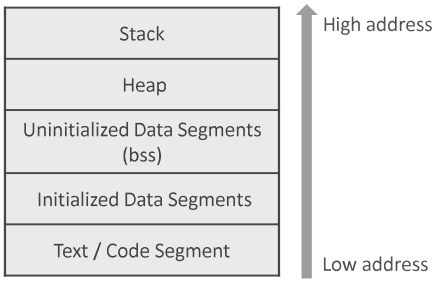
- Text segment contains machine code of the compiled program.
- The text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors.
- The text segment is often read-only, to prevent a program from accidentally modifying its instructions.



Memory Layout of C Programs

2) Initialized Data Segment

- Usually called simply the Data Segment.
- Stores all global and static variables that are initialized by the programmer.

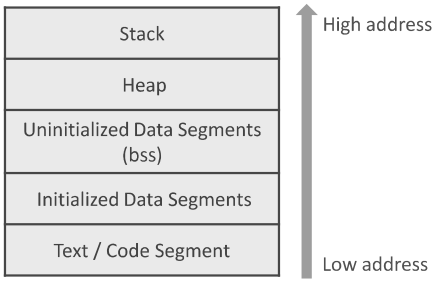


electronics010.blogspot.com C language Eng. Mohamed Yousef

Memory Layout of C Programs

3) Uninitialized Data Segment

- Usually often called the "bss" (block started by symbol) segment.
- contains all global variables and static variables that are do not have explicit initialization in source code.
- Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.

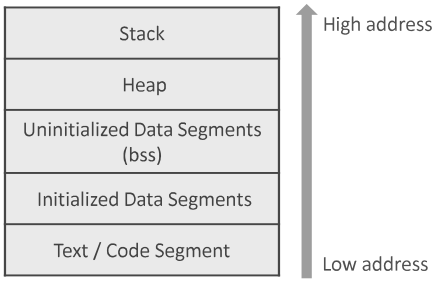


electronics010.blogspot.com C language Eng. Mohamed Yousef

Memory Layout of C Programs

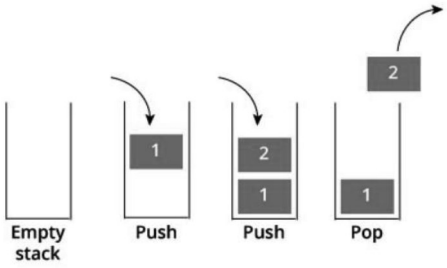
4) Heap

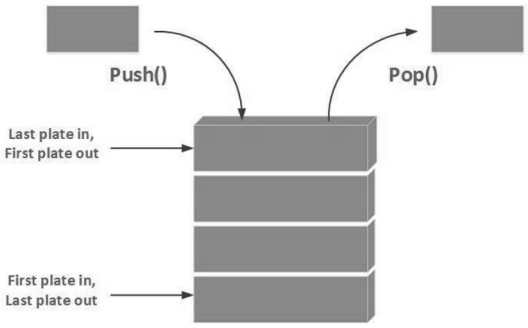
- Heap is the segment where dynamic memory allocation usually takes place.
- Heap area is managed by malloc, realloc, and free.



Function Call Stack and Stack Frames

- To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a ***stack***.
- Think of a stack as analogous to a pile of dishes.
- When a dish is placed on the pile, it's normally placed at the top (referred to as ***pushing*** the dish onto the stack).
- Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as ***popping*** the dish off the stack).
- Stacks are known as last-in, first-out (***LIFO***) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

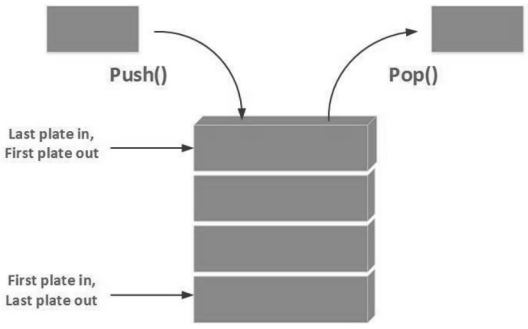




electronics010.blogspot.com C language Eng. Mohamed Yousef

Function Call Stack and Stack Frames

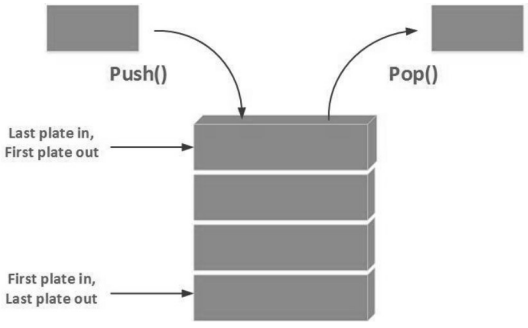
- Each time a function calls another function, an entry is pushed onto the stack.
- This entry, called a ***stack frame***, contains the return address that the called function needs in order to return to the calling function.
- Most functions have automatic variables — parameters and some or all of their local variables.
- The called function's stack frame is a perfect place to reserve the memory for automatic variables.
- That stack frame exists only as long as the called function is active.
- When that function returns — and no longer needs its local automatic variables — its stack frame is popped from the stack, and those local automatic variables are no longer known to the program.



electronics010.blogspot.com C language Eng. Mohamed Yousef

Function Call Stack and Stack Frames

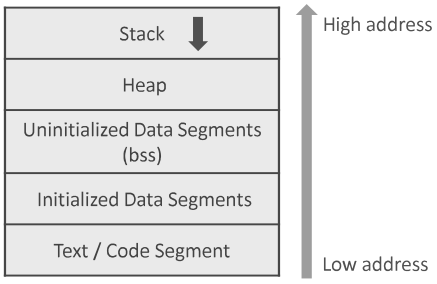
- Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function call stack.
- If more function calls occur than can have their stack frames stored on the function call stack, a fatal error known as ***stack overflow*** occurs.



Memory Layout of C Programs

5) Stack

- The stack holds local (automatic) variables, temporary information, function parameters, return address.
- Typically the stack grows downward, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap.
- when the stack pointer met the heap pointer, free memory was exhausted (stack overflow error).
- This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

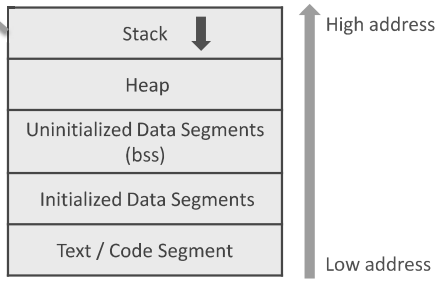


electronics010.blogspot.com C language Eng. Mohamed Yousef

Memory Layout of C Programs

```
int function1(int x){
    int y = 5;
    return(x*y);
}
```

function1 frame	
return address	
x	passed value
y	5



electronics010.blogspot.com C language Eng. Mohamed Yousef

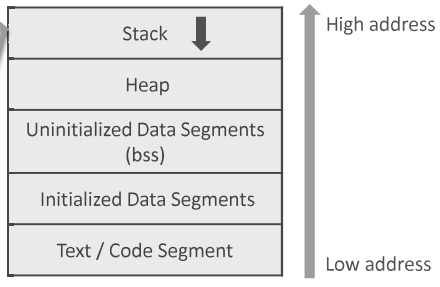
Memory Layout of C Programs

```
int function1(int x){
    int y = 5;
    return(x*y);
}
```

function1 frame	
return address	
x	passed value
y	5

```
int function2(int x){
    int y = 5;
    return(function1(x));
}
```

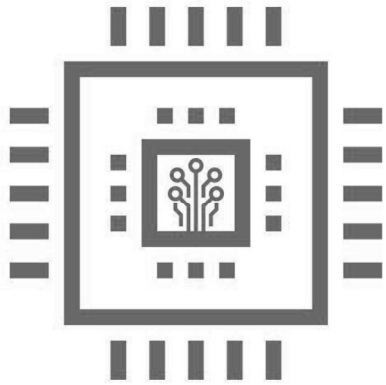
function2 frame	
return address	
x	passed value
y	5



لا يوجد علاقة بين المتغيرات x – y الموجودة فى function1 والمتغيرات x – y الموجودة فى function2، لأن كل دالة لها frame خاص بها فى الـ stack

Embedded Systems

C language



11- Storage classes

Eng. Mohamed Yousef

Storage classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.

These specifiers precede the type that they modify.

```
storage-class type identifier
```

We have four different storage classes:

1) auto
2) register
3) extern
4) static

electronics010.blogspot.com C language Eng. Mohamed Yousef

auto

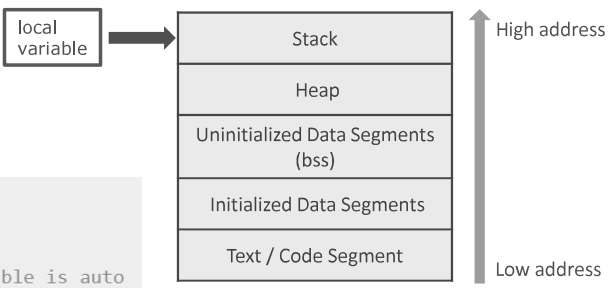
The auto storage-class specifier declares an automatic (local) variable.
This is the default storage class for variables declared inside a function or a block.
By default every local variable is declared as `auto`, hence adding `auto` keyword is pointless.

Syntax to declare `auto` variable

```
auto data_type variable_name;
```

Example to declare `auto` variable

```
int main()
{
    auto int num;
    float amount; // By default every local variable is auto

    return 0;
}
```



register

➢ The register storage class is used to define local variables that should be stored in a register of the microprocessor if a free register is available instead of RAM.
➢ If a free register is not available, then stored in the memory only.
➢ This makes the use of register variables to be much faster than that of the variables stored in the memory.
➢ In addition, we cannot store all our variables in registers because registers size is comparably very small to that of RAM.
➢ **An important** and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

Syntax to declare `register` variable

```
register data_type variable_name;
```

Example to declare `register` variable

```
register int counter;
register int i;
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

register

Doesn't usually make sense in embedded microcontroller system...

On a microcontroller where the CPU and memory are integrated in the same package, the access times associated with retrieving data from RAM are generally very fast, so declaring a register variable doesn't often make sense.

At the moment this modifier has no special meaning in mikroC PRO for PIC.
mikroC PRO for PIC simply ignores requests for register allocation.

electronics010.blogspot.com C language Eng. Mohamed Yousef

main.c

```c
#include <stdio.h>

void Fun_File1(void);

int main(){
    Fun_File1();
    return 0;
}

void Fun_File1(void){

   printf("\n Hello World\n");
}
```

```
Hello World
```

main.c

```c
#include <stdio.h>

void Fun_File1(void);

int main(){
    Fun_File1();
    return 0;
}

void Fun_File1(void){

   printf("\n Hello World\n");
}
```

```
Hello World
```

It is common to divide a big C program among various C files for easy maintenance and modularity.
To share variables and functions among all these files we declare variables and functions in one C file, and use them in other C files through extern modifier.

main.c

```c
extern void Fun_File1(void);

int main(){
    Fun_File1();
    return 0;
}
```

File1.c

```c
#include <stdio.h>

void Fun_File1(void){

   printf("\n Hello World\n");
}
```

```
Hello World
```


main.c

```c
void Fun_File1(void);

int main(){
    Fun_File1();
    return 0;
}
```

File1.c

```c
#include <stdio.h>

void Fun_File1(void){

   printf("\n Hello World\n");
}
```

```
Hello World
```

All *function declarations* are considered as *"extern" by default*, so there is no need to specify it explicitly.

main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);

int main(){
    printf("\n global_var before call = %d\n", global_var);
    Fun_File1();
    printf("\n global_var after  call = %d\n", global_var);

    return 0;
}

void Fun_File1(void){

   global_var++;
}
```

```
global_var before call = 5

global_var after  call = 6
```


main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);

int main(){
    printf("\n global_var before call = %d\n", global_var);
    Fun_File1();
    printf("\n global_var after  call = %d\n", global_var);

    return 0;
}
```

File1.c

```c
void Fun_File1(void){

   global_var++;
}
```

```
error: 'global_var' undeclared (first use in this function)
```


main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);

int main(){
    printf("\n global_var before call = %d\n", global_var);
    Fun_File1();
    printf("\n global_var after  call = %d\n", global_var);

    return 0;
}
```

File1.c

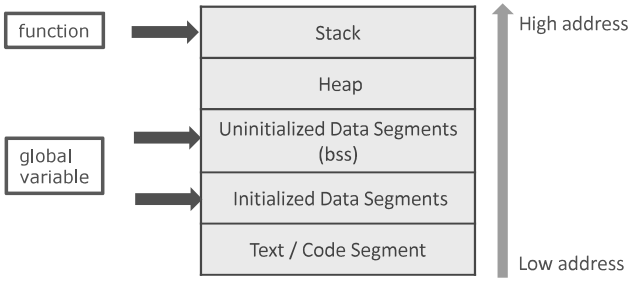
declaration →

```c
extern int global_var;

void Fun_File1(void){

   global_var++;
}
```

```
global_var before call = 5

global_var after  call = 6
```

extern

We use extern (external) storage class to define reference to a global variable or function defined in some other C program.



main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);

int main(){
    printf("\n global_var before call = %d\n", global_var);
    Fun_File1();
    printf("\n global_var after  call = %d\n", global_var);

    return 0;
}
```

assignment

declaration

File1.c

```c
extern int global_var;

void Fun_File1(void){

   global_var++;
}
```

File2.c

```c
int global_var = 10;

void Fun_File2(void){

   global_var++;
}
```

assignment

```
error: ld returned 1 exit status
=== Build failed: 1 error(s), 0 warning(s)
```


main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);
void Fun_File2(void);

int main(){
    printf("\n global_var before call = %d\n", global_var);
    Fun_File1();
    printf("\n global_var after  Fun_File1 = %d\n", global_var);
    Fun_File2();
    printf("\n global_var after  Fun_File2 = %d\n", global_var);

    return 0;
}
```

The static modifier may be applied to global variables.
When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

File1.c

```c
extern int global_var;

void Fun_File1(void){

   global_var++;
}
```

File2.c

```c
static int global_var = 10;

void Fun_File2(void){

   global_var++;
}
```

main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);
void Fun_File2(void);

int main(){
    printf("\n global_var before call = %d\n", global_var);
    Fun_File1();
    printf("\n global_var after  Fun_File1 = %d\n", global_var);
    Fun_File2();
    printf("\n global_var after  Fun_File2 = %d\n", global_var);

    return 0;
}
```

```
global_var before call = 5

global_var after  Fun_File1 = 6

global_var after  Fun_File2 = 6
```

File1.c

```c
extern int global_var;

void Fun_File1(void){

   global_var++;
}
```

File2.c

```c
static int global_var = 10;

void Fun_File2(void){

   global_var++;
}
```

The static modifier may be applied to global variables.
When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

electronics010.blogspot.com C language Eng. Mohamed Yousef

main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);

int main(){
    Fun_File1();

    return 0;
}
```

File1.c

```c
extern int global_var;

void Fun_File1(void){

   global_var++;
}
```

File2.c

```c
#include <stdio.h>

void Fun_File1(void){

   printf("Hello");
}
```

```
multiple definition of `Fun_File1'
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

main.c

```c
#include <stdio.h>

int global_var = 5;

void Fun_File1(void);

int main(){
    Fun_File1();

    return 0;
}
```

File1.c

```c
extern int global_var;

void Fun_File1(void){

   global_var++;
}
```

File2.c

```c
#include <stdio.h>

static void Fun_File1(void){

   printf("Hello");
}
```

In C, functions are global by default. The "static" keyword before a function name makes it static.
Unlike global functions in C, access to static functions is restricted to the file where they are declared.
Another reason for making functions static can be reuse of the same function name in other files.

```c
#include <stdio.h>

void Fun_Static(void);

int main(){
    Fun_Static();
    Fun_Static();
    Fun_Static();

    return 0;
}

void Fun_Static(void){
int local_var = 5;
static int static_var = 0;

    local_var++;
    static_var++;

    printf("\n local_var = %d \t static_var = %d", local_var, static_var);
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

void Fun_Static(void);

int main(){
    Fun_Static();
    Fun_Static();
    Fun_Static();

    return 0;
}

void Fun_Static(void){
int local_var = 5;
static int static_var = 0;

    local_var++;
    static_var++;

    printf("\n local_var = %d \t static_var = %d", local_var, static_var);
}
```

```
local_var = 6   static_var = 1
local_var = 6   static_var = 2
local_var = 6   static_var = 3
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>

void Fun_Static(void);

int main(){
    Fun_Static();
    Fun_Static();
    Fun_Static();

    return 0;
}

void Fun_Static(void){
int local_var = 5;
static int static_var = 0;

    local_var++;
    static_var++;

    printf("\n local_var = %d \t static_var = %d", local_var, static_var);
}
```

```
local_var = 6   static_var = 1
local_var = 6   static_var = 2
local_var = 6   static_var = 3
```

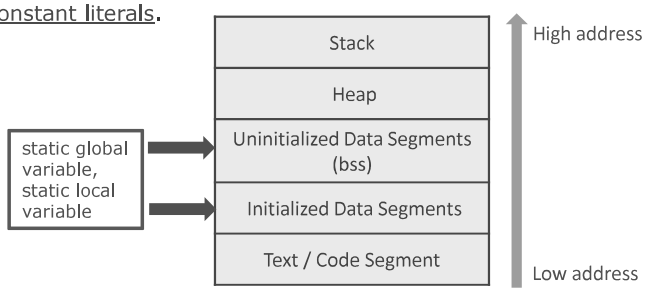
The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.
Therefore, making local variables static allows them to maintain their values between function calls.

Static

- Static variables are allocated within data segment of the program instead of C stack.
- Static variable persists, even after end of function or block.
- Static variables (like global variables) are initialized as 0 if not initialized explicitly.
- You can access a global variable outside the program. However, you cannot access a global static variable outside the program.
- In C, static variables can only be initialized using constant literals.

```
void func(void){
int i = 5;
static int x = i;

    printf(" x = %d", x);
}
```

```
error: initializer element is not constant
```



Syntax to declare static variable

```
static data_type var_name = var_value;
```


LAB

Output of following program?

```
#include <stdio.h>
int main()
{
    static int i=5;
    if(--i){
        main();
        printf("%d ",i);
    }
}
```

A 4 3 2 1

B 1 2 3 4

C 0 0 0 0

D Compiler Error

Output of following program?

```
#include <stdio.h>
int main()
{
    static int i=5;
    if(--i){
        main();
        printf("%d ",i);
    }
}
```

A 4 3 2 1

B 1 2 3 4

✔ 0 0 0 0

D Compiler Error

Question 2 Explanation:
A static variable is shared among all calls of a function. All calls to main() in the given program share the same i. i becomes 0 before the printf() statement in all calls to main().

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
#include <stdio.h>
int main()
{
    static int i=5;
    if (--i){
        printf("%d ",i);
        main();
    }
}
```

A 4 3 2 1

B 1 2 3 4

C 4 4 4 4

D 0 0 0 0

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
#include <stdio.h>
int main()
{
    static int i=5;
    if (--i){
        printf("%d ",i);
        main();
    }
}
```

✔ 4 3 2 1

B 1 2 3 4

C 4 4 4 4

D 0 0 0 0

Question 3 Explanation:
Since i is static variable, it is shared among all calls to main(). So is reduced by 1 by every function call.

```
#include <stdio.h>
int main()
{
  int x = 10;
  static int y = x;

  if(x == y)
     printf("Equal");
  else if(x > y)
     printf("Greater");
  else
     printf("Less");
  return 0;
}
```

A Compiler Error

B Equal

C Greater

D Less

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
#include <stdio.h>
int main()
{
  int x = 10;
  static int y = x;

  if(x == y)
     printf("Equal");
  else if(x > y)
     printf("Greater");
  else
     printf("Less");
  return 0;
}
```

✓ Compiler Error

B Equal

C Greater

D Less

Question 10 Explanation:

In C, static variables can only be initialized using constant literals. This is allowed in C++ though. See this GFact for details.

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
int f(int n)
{
   static int i = 1;
   if (n >= 5)
      return n;
   n = n+i;
   i++;
   return f(n);
}
```

The value returned by f(1) is

A 5

B 6

C 7

D 8

```
int f(int n)
{
   static int i = 1;
   if (n >= 5)
      return n;
   n = n+i;
   i++;
   return f(n);
}
```

The value returned by f(1) is

A 5

B 6

✓ 7

D 8

Question 11 Explanation:

Since i is static, first line of f() is executed only once.

```
Execution of f(1)
    i = 1
    n = 2
    i = 2
 Call f(2)
    i = 2
    n = 4
    i = 3
 Call f(4)
   i = 3
   n = 7
   i = 4
 Call f(7)
  since n >= 5 return n(7)
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

In C, static storage class cannot be used with:

A Global variable

B Function parameter

C Function name

D Local variable

electronics010.blogspot.com C language Eng. Mohamed Yousef

In C, static storage class cannot be used with:

A Global variable

✓ Function parameter

C Function name

D Local variable

Question 12 Explanation:

Declaring a global variable as static limits its scope to the same file in which it is defined. A static function is only accessible to the same file in which it is defined. A local variable declared as static preserves the value of the variable between the function calls.

```c
#include <stdio.h>
int a, b, c = 0;
void prtFun (void);
int main ()
{
    static int a = 1; /* line 1 */
    prtFun();
    a += 1;
    prtFun();
    printf ( "n %d %d " , a, b) ;
}

void prtFun (void)
{
    static int a = 2; /* line 2 */
    int b = 1;
    a += ++b;
    printf (" n %d %d " , a, b);
}
```

A
3 1
4 1
4 2

B
4 2
6 1
6 1

C
4 2
6 2
2 0

D
3 1
5 2
5 2

electronics010.blogspot.com C language Eng. Mohamed Yousef


```c
#include <stdio.h>
int a, b, c = 0;
void prtFun (void);
int main ()
{
    static int a = 1; /* line 1 */
    prtFun();
    a += 1;
    prtFun();
    printf ( "n %d %d " , a, b) ;
}

void prtFun (void)
{
    static int a = 2; /* line 2 */
    int b = 1;
    a += ++b;
    printf (" n %d %d " , a, b);
}
```

A
3 1
4 1
4 2

B
4 2
6 1
6 1

✔
4 2
6 2
2 0

D
3 1
5 2
5 2

main	prtFun	Output
a = 1	a = 2 b = 1 a = a + (++b) a = 2 + 2 = 4 b = 2 print a , b	4 2
a += 1 a = a + 1 a = 1 + 1 = 2	a = 4 b = 1 a = a + (++b) a = 4 + 2 = 6 b = 2 print a , b	6 2
print a , b		2 0

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
#include <stdio.h>
int main()
{
  extern int i;
  printf("%d ", i);
  {
       int i = 10;
       printf("%d ", i);
  }
}
```

A 0 10

B Compiler Error

C 0 0

D 10 10

```
#include <stdio.h>
int main()
{
  extern int i;
  printf("%d ", i);
  {
       int i = 10;
       printf("%d ", i);
  }
}
```

A 0 10

✓ Compiler Error

C 0 0

D 10 10

There is no global definition for i

Which of the followings is correct for a function definition along with storage-class specifier in C language?

A int fun(auto int arg)

B int fun(static int arg)

C int fun(register int arg)

D int fun(extern int arg)

E All of the above are correct.

Which of the followings is correct for a function definition along with storage-class specifier in C language?

A int fun(auto int arg)

B int fun(static int arg)

✓ int fun(register int arg)

D int fun(extern int arg)

E All of the above are correct.

Question 23 Explanation:

As per C standard, "*The only storage-class specifier that shall occur in a parameter declaration is register.*" That's why correct answer is C.

Which of the following storage classes have global visibility in C/C++ ?

A Auto

B Extern

C Static

D Register

electronics010.blogspot.com C language Eng. Mohamed Yousef

Which of the following storage classes have global visibility in C/C++ ?

A Auto

✓ Extern

C Static

D Register

electronics010.blogspot.com C language Eng. Mohamed Yousef

The following program

```
main()
{
inc(); inc(); inc();
}
inc()
{
static int x;
printf("%d", ++x);
}
```

A prints 012

B prints 123

C prints 3 consecutive, but unpredictable numbers

D prints 111

The following program

```
main()
{
inc(); inc(); inc();
}
inc()
{
static int x;
printf("%d", ++x);
}
```

A prints 012

✔ prints 123

C prints 3 consecutive, but unpredictable numbers

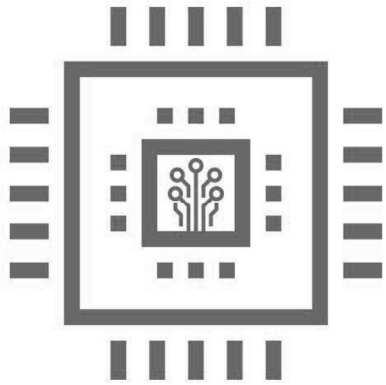
D prints 111

Question 28 Explanation:

Since the value of x is not declared it would automatically get 0 as its initial value. First time when inc() is called in main(), the value of x will be incremented by 1. Static variables preserve their previous value in their previous scope and are not initialized again in the new scope. So when inc() is called second and third time, the value of x is simply incremented by 1. That's why the correct answer is 123

Embedded Systems

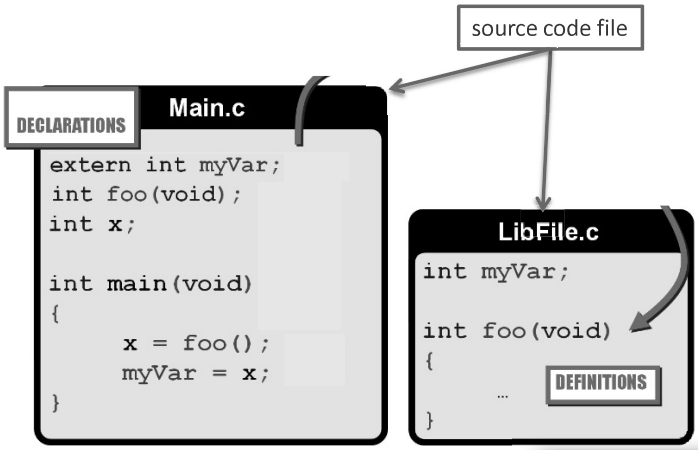
C language



12- Building Larger Programs

Eng. Mohamed Yousef

Building Larger Programs



Header File

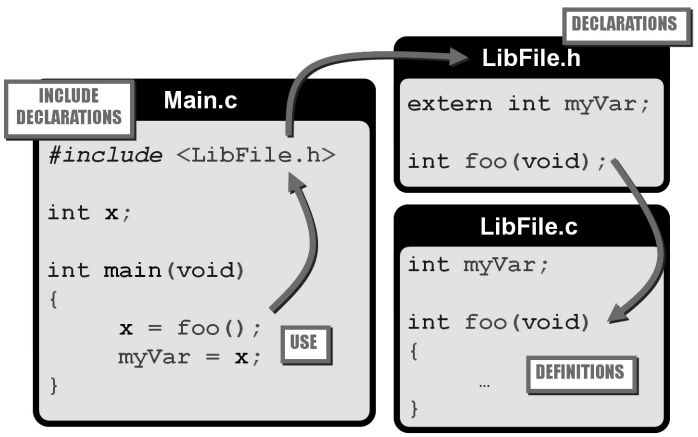
- As multi-module projects grow more complex, you find the first part of each source code file growing longer and longer.
- More prototypes, more constants, and more global variables and structures are required for each module.
- Rather than burden your code with redundancies, you can create a ***header file*** for the project.
- Header files end with an .h extension and contain function prototypes including various data types and/or constants required by the functions.

What's a module?

A *module* is a source code file and its compiled object file. Together, the source code and object files are what I call a module.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Building Larger Programs



Header File

- As multi-module projects grow more complex, you find the first part of each source code file growing longer and longer.
- More prototypes, more constants, and more global variables and structures are required for each module.
- Rather than burden your code with redundancies, you can create a ***header file*** for the project.
- Header files end with an .h extension and contain function prototypes including various data types and/or constants required by the functions.

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this –

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

This construct is commonly known as a wrapper **#ifndef**. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

electronics010.blogspot.com C language Eng. Mohamed Yousef

code1

code2

```
/*
================================================================================
==
PROGRAM     : single file project
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/
#include <stdio.h>

// variables
int operation = 0;
float operand1 = 0, operand2 = 0;

// constants
#define ADD 1
#define SUB 2

// prototypes
void Title(void);
void Get_User_Inputs(void);
void Add(void);
void Sub(void);

//
--------------------------------------------------------------------------------
int main(){

    Title();
    Get_User_Inputs();

    switch(operation){
        case ADD: Add(); break;
        case SUB: Sub(); break;
        default: printf("\n Tanks");
    }

    return 0;
}
//
--------------------------------------------------------------------------------
void Title(void){
    printf("\n\t\tCalculator Program");
    printf("\n\t\t==================\n");
}
//
--------------------------------------------------------------------------------
void Get_User_Inputs(void){

    printf("\n 1- Addition.");
    printf("\n 2- Subtraction.");
    printf("\n Select Operation :");
    scanf("%d", &operation);

    printf("\n Enter 1st operand :");
    scanf("%f", &operand1);
```

```
    printf(" Enter 2nd operand :");
    scanf("%f", &operand2);
}
//
--------------------------------------------------------------------------
void Add(void){
     printf("\n %g + %g = %g", operand1, operand2, (operand1 + operand2));
}
//
--------------------------------------------------------------------------
void Sub(void){
     printf("\n %g - %g = %g", operand1, operand2, (operand1 - operand2));
}
//
--------------------------------------------------------------------------
```

```
/*
================================================================================
==
PROGRAM     : multi files project
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/
#include "calculator.h"

//
--------------------------------------------------------------------------
int main(){
    Calculator();
    return 0;
}
```

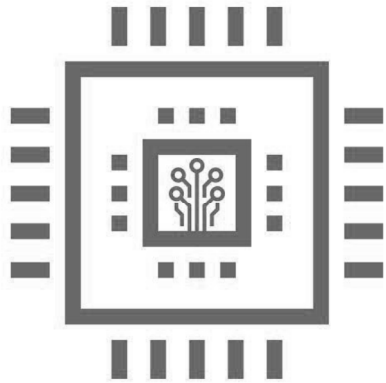
```
#ifndef CALCULATOR_H_INCLUDED
#define CALCULATOR_H_INCLUDED

extern void Calculator(void);

#endif // CALCULATOR_H_INCLUDED
```

```
/*
================================================================================
==
PROGRAM     : multi files project
Author      : Mohamed Sayed Yousef
              http://electronics010.blogspot.com.eg/
Date        : September 2018
Version     : 1.0
Description :
================================================================================
==
*/
#include <stdio.h>

// variables
static  int   operation = 0;
static  float operand1  = 0, operand2 = 0;

// constants
#define ADD 1
#define SUB 2

// prototypes
void Calculator(void);
static void Title(void);
static void Get_User_Inputs(void);
static void Add(void);
static void Sub(void);

//
--------------------------------------------------------------------------
void Calculator(void){
    Title();
    Get_User_Inputs();

    switch(operation){
        case ADD: Add(); break;
        case SUB: Sub(); break;
        default: printf("\n Tanks");
    }
}
//
--------------------------------------------------------------------------
static void Title(void){
    printf("\n\t\tCalculator Program");
    printf("\n\t\t==================\n");
}
//
--------------------------------------------------------------------------
static void Get_User_Inputs(void){

    printf("\n 1- Addition.");
    printf("\n 2- Subtraction.");
    printf("\n Select Operation :");
    scanf("%d", &operation);

    printf("\n Enter 1st operand :");
    scanf("%f", &operand1);

    printf(" Enter 2nd operand :");
```

```
    scanf("%f", &operand2);
}
//
-------------------------------------------------------------------------
static void Add(void){
     printf("\n %g + %g = %g", operand1, operand2, (operand1 + operand2));
}
//
-------------------------------------------------------------------------
static void Sub(void){
     printf("\n %g - %g = %g", operand1, operand2, (operand1 - operand2));
}
//
-------------------------------------------------------------------------
```

Embedded Systems

C language



13- Arrays

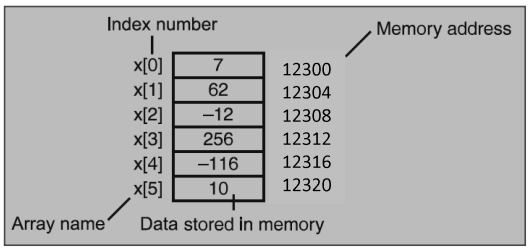
Eng. Mohamed Yousef

Arrays

An array is collection of items stored at continuous memory locations.

Arrays share a few common attributes:

- Variables in an array share the same name
- Variables in an array share the same data type
- Individual variable in an array are called element
- Elements in an array are accessed with an index number
- Elements in an array begin with index number zero



electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

How to Create an Array?

Syntax

```
type arrayName [size];
```

- size refers to the number of elements
- size must be a constant integer

Example

```
1  int a[10];    // An array that can hold 10 integers
2
3  char s[25];   // An array that can hold 25 characters
```

Arrays

How to Initialize an Array at Declaration?

Syntax

```
type arrayName [size] = {item1, …, itemn};
```

The items must all match the type of the array

Example

```
1  int a[5] = {10, 20, 30, 40, 50};
2
3  char b[5] = {'a', 'b', 'c', 'd', 'e'};
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

How to Initialize an Array at Declaration?

You can quickly initialize your arrays with a single default value as demonstrated in the following array declaration.

```
int iArray[5] = {0};
```

Assigning the single numeric value of 0 in an array declaration will, by default, assign all array elements the value of 0.

```
int highscore[] = { 750, 699, 675 };
```

The number in the square brackets isn't necessary when you initialize an array, as shown in the preceding example. That's because the compiler is smart enough to count the elements and configure the array automatically.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

How to Use an Array?

Arrays are accessed like variables, but with an index.

```
unsigned int Total[5];
Total[1] = 25;
Temp = Total[2];
```

Arrays

```c
#include <stdio.h>

int main(){
    int x;    int iArray[5];

    //initialize array elements
    for ( x = 0; x < 5; x++ ){
        iArray[x] = x + 5;
    }

    //print array element
    for ( x = 0; x < 5; x++ ){
        printf("\n iArray[%d] = %d", x, iArray[x]);
    }

    return 0;
}
```

```
iArray[0] = 5
iArray[1] = 6
iArray[2] = 7
iArray[3] = 8
iArray[4] = 9
```


Arrays

```c
/*
 * C program to find average of marks using array
 */

#include <stdio.h>

#define SIZE 5    // Size of the array

int main(){
    int marks[SIZE] = {0}; // Declare an array of size 5
    int index = 0, sum = 0;
    float avg = 0;

    printf("\n Enter marks of %d students: ", SIZE);

    // Input marks of all students in marks array
    for(index = 0; index < SIZE; index++){
        scanf("%d", &marks[index]);
    }

    // Find sum of all marks
    for(index = 0; index < SIZE; index++){
        sum += marks[index];
    }

    // Calculate average of marks
    avg = (float) sum / SIZE;

    // Print the average marks
    printf(" Average marks = %g", avg);

    return 0;
}
```

```
Enter marks of 5 students: 11
14
15
9
13

Average mark = 12.4
```


Arrays

What happens if you have an array with ten elements and you accidently write to the eleventh element? C has no bounds checking for array indexes. Therefore you may read or write to an element not declared in the array, however this will generally have disastrous results. Often this will cause the program to crash

C does not allow you to assign the value of one array to another simply by using an assignment like:

```
int a[10],b[10];
.
.
a=b;
```

The above example is wrong. If you want to copy the contents of one array into another, you must copy each individual element from the first array into the second array. The following example shows how to copy the array `a[ ]` into `b[ ]` assuming that each array has 20 elements.

```
for(i=0;i<20;i++)
    b[i] = a[i];
```

Arrays

Strings

Strings are arrays of `char` whose last element is a null character '\0' with an ASCII value of 0. C has no native string data type, so strings must always be treated as character arrays.

Strings:

- Are enclosed in double quotes `"string"`
- Are terminated by a null character `'\0'`
- Must be manipulated as arrays of characters (treated element by element)
- May be initialized with a string literal

```
/* The two declarations are identical: */
char msg1[5] = {'T', 'e', 's', 't', '\0'};
char msg2[] = "Test";
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

```
/*
 * C program to displaying a text
 */

#include <stdio.h>

int main(){
char sentence[] = "Random text";
int index = 0;

    while(sentence[index] != '\0'){
        printf("%c", sentence[index]);
        index++;
    }

    return 0;
}
```

```
Random text
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

```
/*
 * C program to displaying a text
 */

#include <stdio.h>

int main(){
char sentence[] = "Random text";

    printf("%s", sentence);

    return 0;
}
```

```
Random text
```

Specifier & Output
s
String of characters

When the `char` array is used in a function, as shown in the preceding line, the square brackets aren't necessary.

Arrays

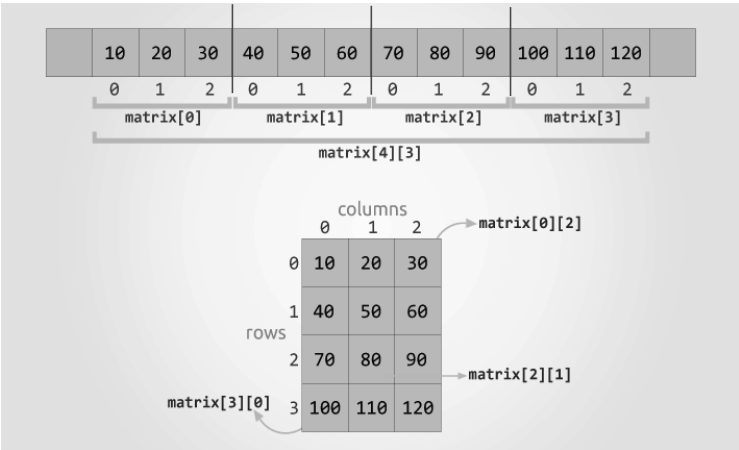
Multi-dimensional array:

Unlike one-dimensional array, multi-dimensional array stores collection of array.

Two-dimensional array

Two-dimensional array is a collection of one-dimensional array.

You can logically represent a two-dimensional array as a matrix.



Arrays

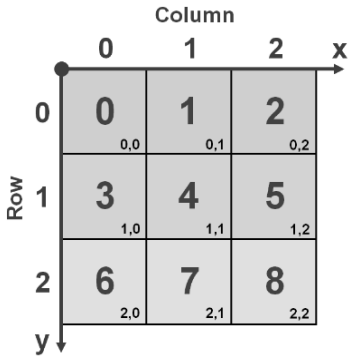
Initializing Multidimensional Arrays at Declaration

```
int a[3][3] = {{0, 1, 2},
               {3, 4, 5},
               {6, 7, 8}};
```

Row, Column

a[y][x]

```
Row 0   a[0][0] = 0;
        a[0][1] = 1;
        a[0][2] = 2;
Row 1   a[1][0] = 3;
        a[1][1] = 4;
        a[1][2] = 5;
Row 2   a[2][0] = 6;
        a[2][1] = 7;
        a[2][2] = 8;
```



Visualizing 2-Dimensional Arrays

Arrays

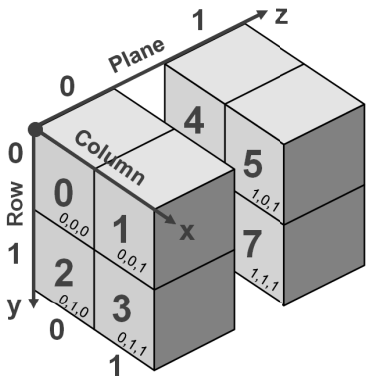
Initializing Multidimensional Arrays at Declaration

```
int a[2][2][2] = {{{0, 1},{2, 3}},
                  {{4, 5},{6, 7}}};
```

Plane, Row, Column

a[z][y][x]

```
Plane 0   a[0][0][0] = 0;
          a[0][0][1] = 1;
          a[0][1][0] = 2;
          a[0][1][1] = 3;
Plane 1   a[1][0][0] = 4;
          a[1][0][1] = 5;
          a[1][1][0] = 6;
          a[1][1][1] = 7;
```



Visualizing 3-Dimensional Arrays

Arrays

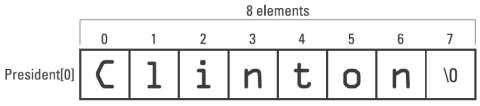
```c
#include <stdio.h>

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

   /* output each array element's value */
   for (i = 0; i < 5; i++){

      for (j = 0; j < 2; j++){

         printf("\n a[%d][%d] = %d", i, j, a[i][j] );
      }
   }
    return 0;
}
```

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

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

```c
/*
  An Array of Strings
*/

#include <stdio.h>
#define NAMES 3
#define LETTERS 8

int main(){
char president[NAMES][LETTERS] = { "Clinton",
                                   "Bush",
                                   "Obama" };
int x = 0, index = 0;

    for(x = 0; x < NAMES; x++){
        index = 0;
        printf("\n ");
        while(president[x][index] != '\0'){

            printf("%c", president[x][index]);
            index++;
        }
    }
    return 0;
}
```

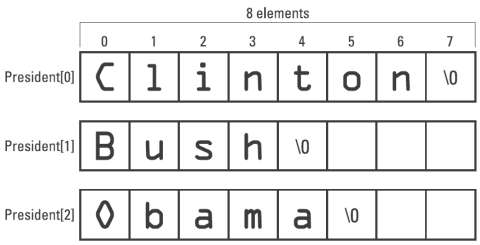



```
Clinton
Bush
Obama
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

```c
/*
  An Array of Strings
*/

#include <stdio.h>
#define NAMES 3


int main(){
char president[NAMES][LETTERS] = { "Clinton",
                                   "Bush",
                                   "Obama" };
int x = 0;

    for(x = 0; x < NAMES; x++){
        printf("\n %s", president[x]);
    }
    return 0;
}
```




```
Clinton
Bush
Obama
```

When working with string elements in an array, the string is referenced by the first dimension only.

Arrays

Passing an array to a function

The function must be prototyped with the array specified as one of the arguments.

It looks like this:

```
void whatever(int nums[]);
```

- The size of the array is not required between the array brackets.
- If it's included, the compiler checks that it's greater than zero, then ignores it.
- Specifying a negative size is a compilation error.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

Passing an array to a function

When you call a function with an array as an argument, you must omit the square brackets:

```
whatever(values);
```

Note:

Arrays are passed by reference by default.

Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their original memory locations..

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

You can pass one element from array to a function.

```
whatever(values[6]);
```

Note:

Element is passed by value.

Arrays

```c
#include <stdio.h>
#define SIZE 5

void showarray(int array[]);

int main(){
    int n[] = {1, 2, 3, 5, 7};

    printf("Here's your array:\n");
    showarray(n);

    return 0;
}

void showarray(int array[]){
    int x = 0;

    for(x = 0; x < SIZE; x++){
        printf("%d\t",array[x]);
    }
}
```

```
Here's your array:
1       2       3       5       7
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

```c
#include <stdio.h>

#define ROWS 3
#define COLS 3

// Function declaration to print two dimensional array
void printMatrix(int mat[][COLS]);

int main(){

    int mat[ROWS][COLS] = {
                            {1, 2, 3},
                            {4, 5, 6},
                            {7, 8, 9}
                          };

    // Print elements of matrix using function
    printMatrix(mat);

    return 0;
}
```

Note:

The function definition specifies the array parameter as `int mat[][3]`.

The first subscript of a multidimensional array is not required, but all subsequent subscripts are required.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Arrays

```c
void printMatrix(int mat[][COLS]){
    int i = 0, j = 0;

    // Print elements of two dimensional array.
    printf("\n Elements in matrix:");

    for (i = 0; i < ROWS; i++){

        printf("\n");

        for (j = 0; j < COLS; j++){
            printf(" %d ", mat[i][j]);
        }
    }
}
```

```
Elements in matrix:
1  2  3
4  5  6
7  8  9
```

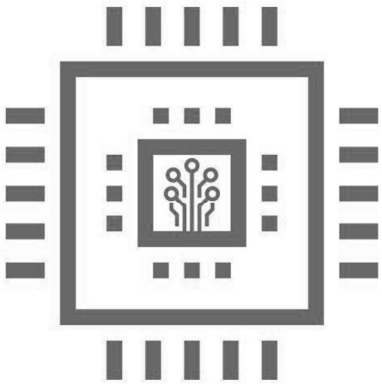
Arrays

Returning an array from a function

A function in C can return an array as pointers.

Embedded Systems

C language



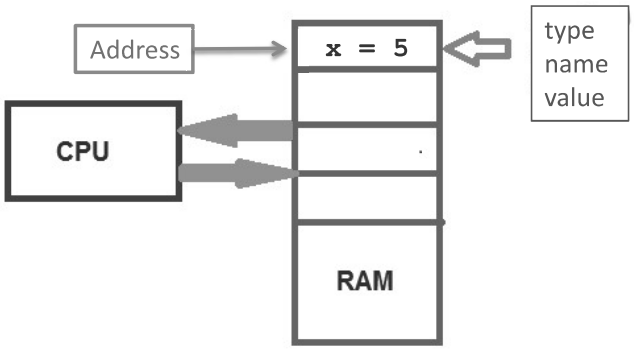
14- Pointers

Eng. Mohamed Yousef

Pointers

```
#include <stdio.h>

int main(){
    int x = 5;

    printf("\n Address of x :%p", &x);

    return 0;
}
```

```
Address of x :0060FF0C
```



p is the conversion specifier to print pointers.

& → address of operator.

electronics010.blogspot.com C language Eng. Mohamed Yousef

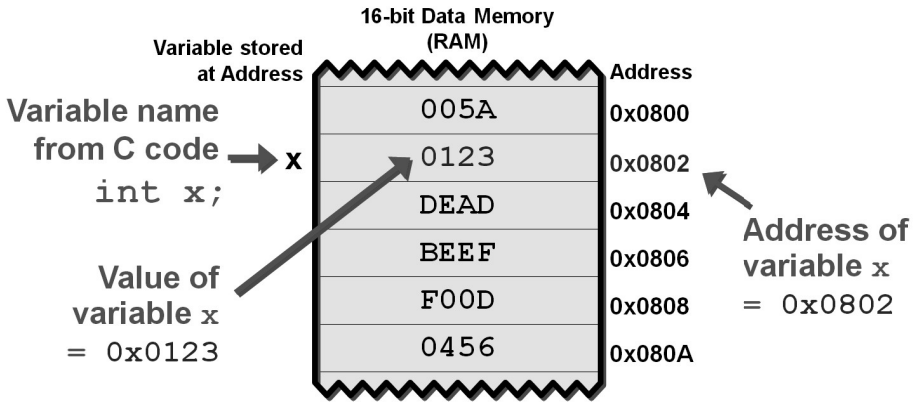
Pointers

```
#include <stdio.h>

int main(){
    int x = 5;
    int * ptr;

    ptr = &x;
    printf("\n Address of x :%p", ptr);

    return 0;
}
```

pointer

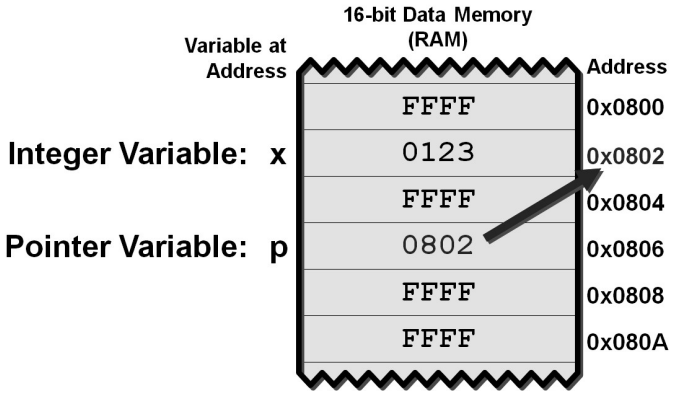
```
Address of x :0060FF08
```

Pointers



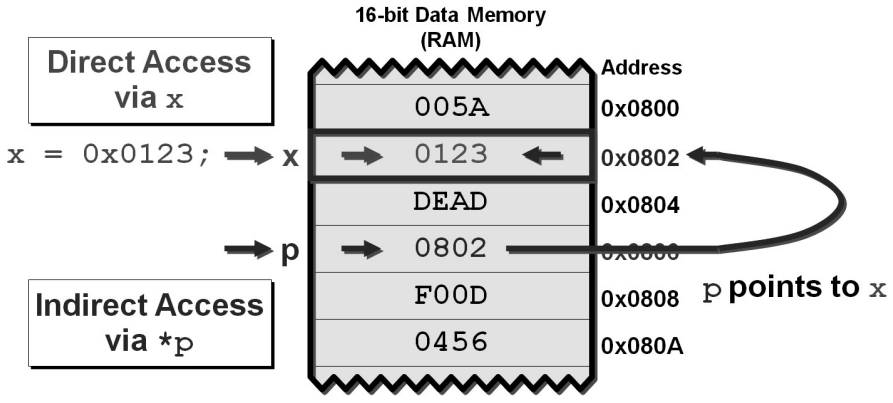
electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers



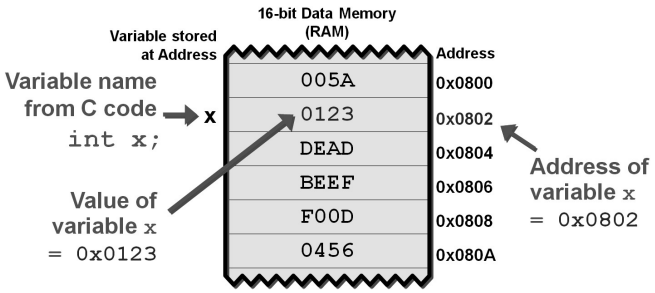
electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers



Pointers

A *pointer* is a variable that contains a memory Address (location).



Syntax

```
type *ptrName;
```

- In the context of a declaration, the * merely indicates that the variable is a pointer.
- *type* is the type of data the pointer may point to.

* → dereference operator

Pointers

Initialization

```
int x = 5;
int * ptr;

ptr = &x;
```

```
int x = 5;
int * ptr = &x;
```


Pointers

Not initializing your pointer variables can result in invalid data or invalid expression results.

Pointer variables should always be initialized

- with another variable's memory address,
- with 0,
- or with the keyword NULL.

```
int *ptr1;
int *ptr2;
int *ptr3;

ptr1 = &x;
ptr2 = 0;
ptr3 = NULL;
```

```
#include <stdio.h>

int main(){
    int * ptr;
    printf("\n Address of x :%p", ptr);

    return 0;
}
```

ERROR

Pointers

- The following code contains a horrible error:

```
#include <stdio.h>

int   main(void)
{
      short  i = 13;
      short  *p;

      *p = 23;
      printf("%hi\n", *p);

      return 0;
}
```

ERROR

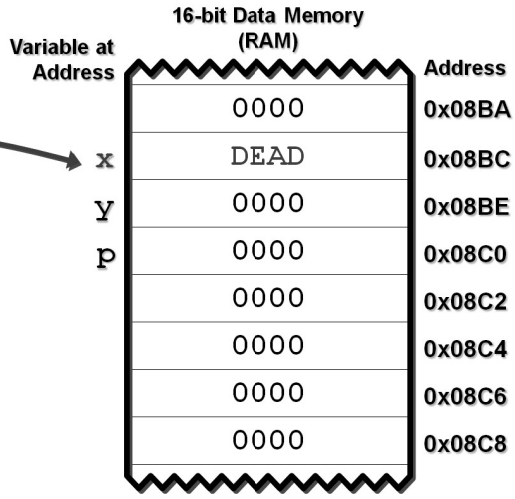
you had just worte on garbage that might be an address

electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers

Assigning Values by Using a Pointer

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
   p = &y;
  *p = 0x0200;
}
```

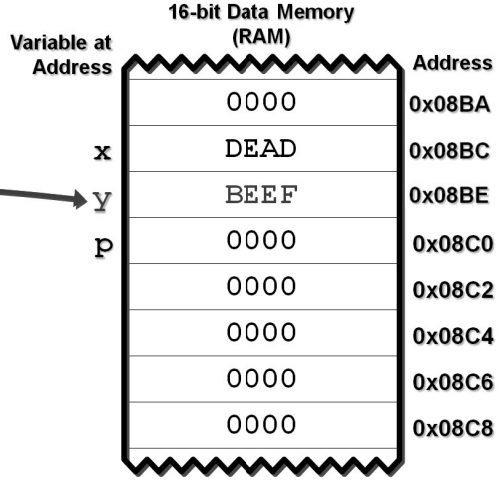


electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers

Assigning Values by Using a Pointer

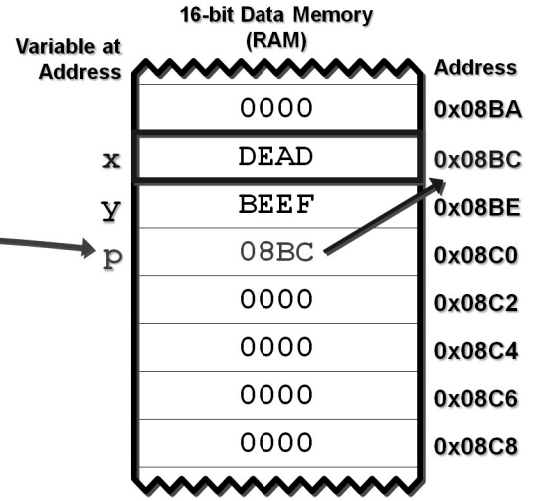
```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
   p = &y;
  *p = 0x0200;
}
```



Pointers

Assigning Values by Using a Pointer

```
{
 int x, y;
 int *p;

 x = 0xDEAD;
 y = 0xBEEF;
 p = &x;

 *p = 0x0100;
  p = &y;
 *p = 0x0200;
}
```

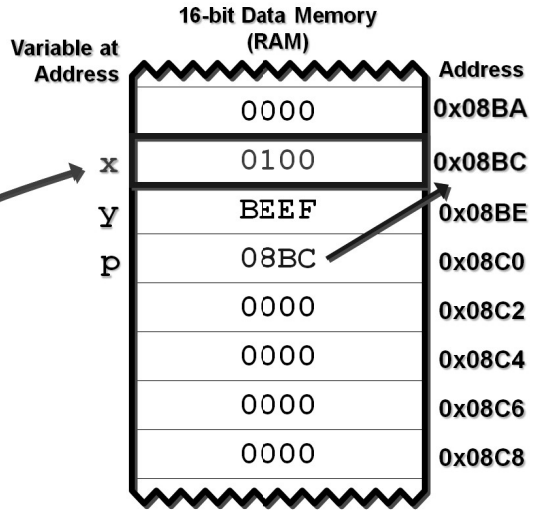


electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers

Assigning Values by Using a Pointer

```
{
 int x, y;
 int *p;

 x = 0xDEAD;
 y = 0xBEEF;
 p = &x;

 *p = 0x0100;
  p = &y;
 *p = 0x0200;
}
```

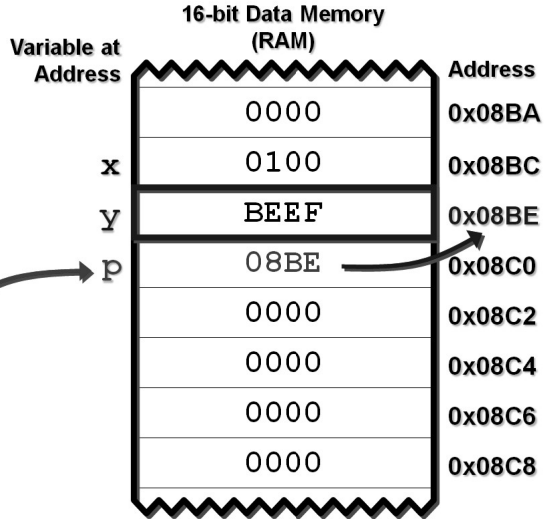


electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers

Assigning Values by Using a Pointer

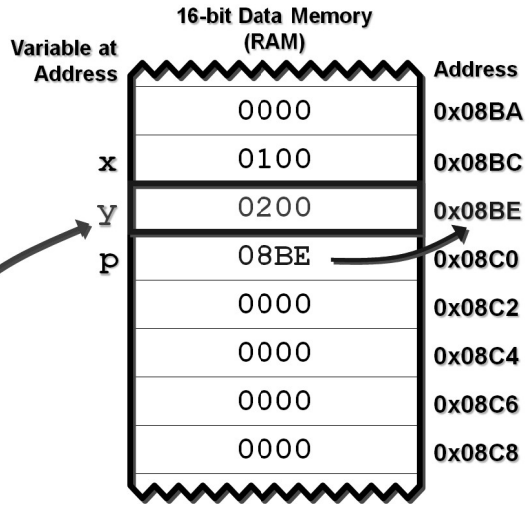
```
{
 int x, y;
 int *p;

 x = 0xDEAD;
 y = 0xBEEF;
 p = &x;

 *p = 0x0100;
  p = &y;
 *p = 0x0200;
}
```



Pointers

Assigning Values by Using a Pointer

```
{
  int x, y;
  int *p;

  x = 0xDEAD;
  y = 0xBEEF;
  p = &x;

  *p = 0x0100;
   p = &y;
  *p = 0x0200;
}
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers

Assigning Values by Using a Pointer

```c
#include <stdio.h>

int main(){
int x = 5;
int *iPtr;

    iPtr = &x;    //iPtr is assigned the address of x
    *iPtr = 7;    //the value of x is indirectly changed to 7

    printf("\n Address of x :%p", &x);
    printf("\n Address of x :%p", iPtr);

    printf("\n Value of x :%d", x);
    printf("\n Value of *iPtr :%d", *iPtr);

    return 0;
}
```

```
Address of x :0060FF08
Address of x :0060FF08
Value of x :7
Value of *iPtr :7
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers and Arrays

```c
#include <stdio.h>

int main(){
int array[5] = {2, 3, 5, 7, 11};

    printf("\n 'array' is at address : %p", &array[0]);
    printf("\n 'array' is at address : %p", &array);
    printf("\n 'array' is at address : %p", array);

    return 0;
}
```

```
'array' is at address : 0060FEFC
'array' is at address : 0060FEFC
'array' is at address : 0060FEFC
```

Array exhibits a special behaviour. Whenever you refer an array name directly, is behaves a pointer pointing at zeroth array element.
Which means both of the below statements are equivalent.

```
int * ptr = &arr[0];
int * ptr = arr;
```

Pointer Arithmetic

A pointer is an address, which is a numeric value.

Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value.

There are four arithmetic operators that can be used on pointers :

- Increment and decrement **++** and **--**
- Addition and Subtraction **+** and **-**

Increment operator when used with a pointer variable, the address stored in a pointer variable is incremented by one unit, not by one digit.

What's a unit?

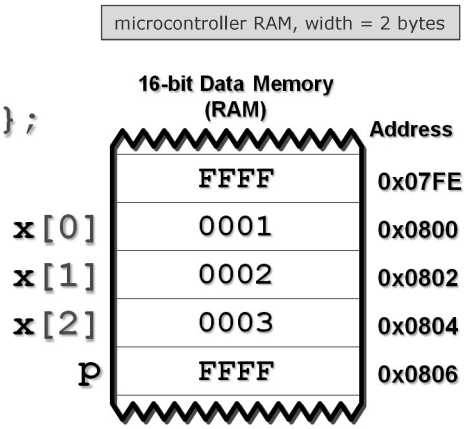
It depends on the variable type. If pointer is a char pointer, pointer incremented by 1 byte.

But if it were an int , on most systems, an int is 4 bytes.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointer Arithmetic

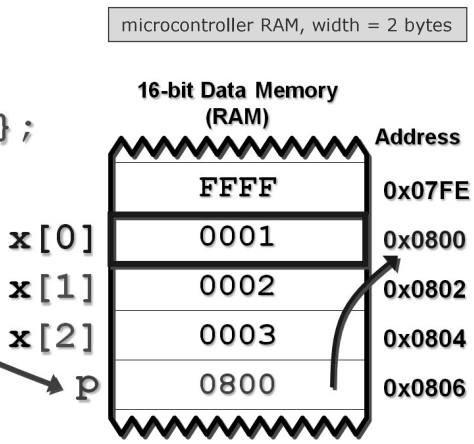
```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointer Arithmetic

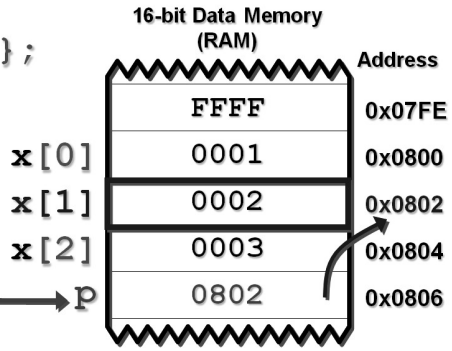
```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```



Pointer Arithmetic

microcontroller RAM, width = 2 bytes

```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```

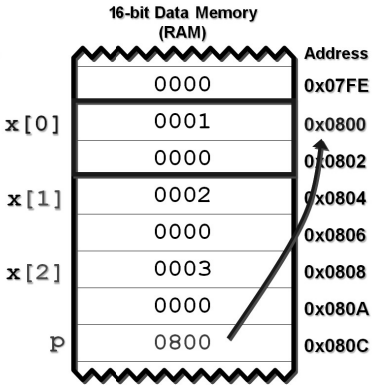


Pointer Arithmetic

Example

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



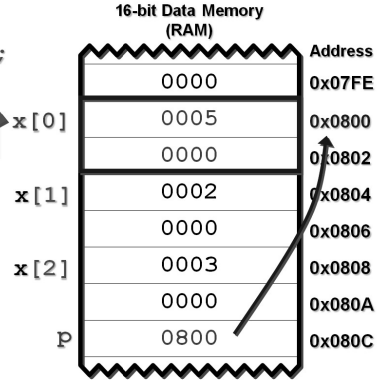
Pointer Arithmetic

Example

microcontroller RAM, width = 2 bytes

```
x[0] = 1:
*p += 4
x[0] += 4
x[0] = x[0] + 4
x[0] = 1 + 4
x[0] = 5
```

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



Pointer Arithmetic

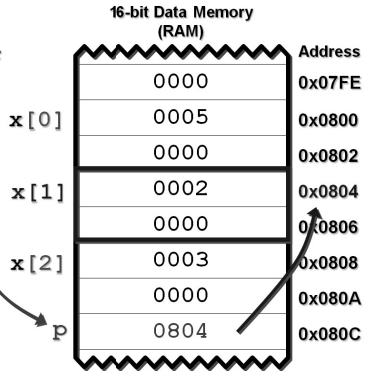
Example

```
p = 0800
p++
p = p + (1 * 4)
p = 0800 + 4
p = 0804
```

4 byte for long type

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



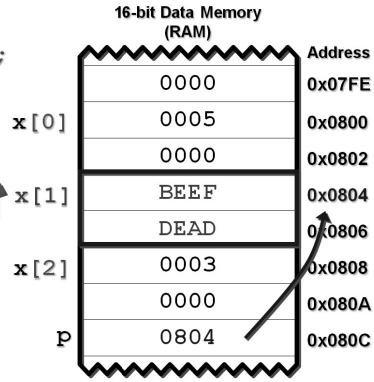
electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointer Arithmetic

Example

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

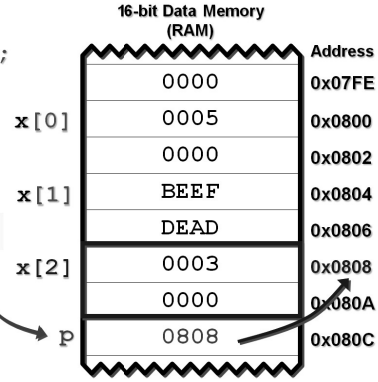
Pointer Arithmetic

Example

Increment the pointer again (by 4 bytes)

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```

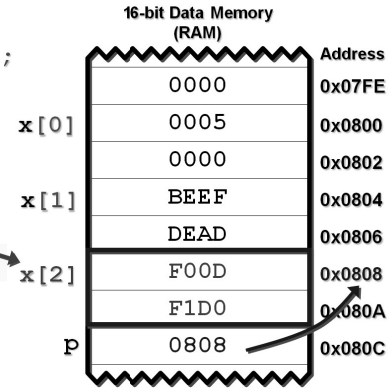


Pointer Arithmetic

Example

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

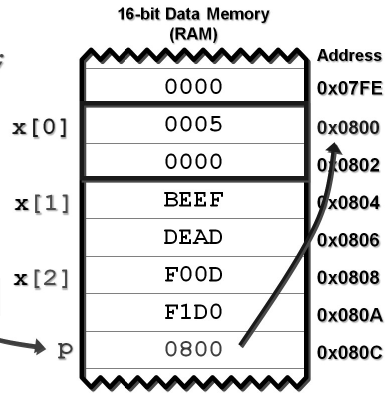
Pointer Arithmetic

Example

```
p = 0808
p -= 2
p = p – (2 * 4)
p = 0808 – 8
p = 0800
```

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



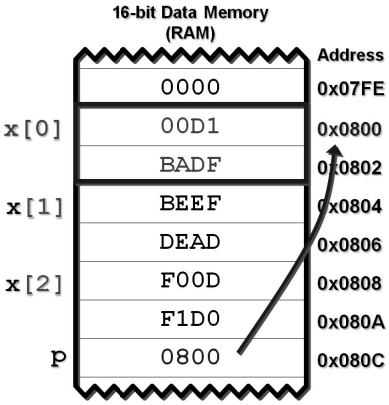
electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointer Arithmetic

Example

microcontroller RAM, width = 2 bytes

```
{
 long x[3] = {1,2,3};
 long *p = &x;

 *p += 4;
 p++;
 *p = 0xDEADBEEF;
 p++;
 *p = 0xF1D0F00D;
 p -= 2;
 *p = 0xBADF00D1;
}
```



Pointer Arithmetic

```
#include <stdio.h>

int main(){
int numbers[5];
int x, *pn;

    pn = numbers; // initialize pointer

    // Fill array
    for(x = 0; x < 5; x++){
        *pn = x + 1;
        pn++;
    }

    // Display array
    for(x = 0; x < 5; x++){
        printf("\n numbers[%d] = %d", x, numbers[x]);
    }

    return 0;
}
```

```
numbers[0] = 1
numbers[1] = 2
numbers[2] = 3
numbers[3] = 4
numbers[4] = 5
```

you should see that each address is separated by 4 bytes (assuming that the size of an int is 4 bytes on your machine).

electronics010.blogspot.com C language Eng. Mohamed Yousef

Post-Increment/Decrement Syntax Rule

Syntax	Operation	Description by Example

*p++
*(p++)
	Post-Increment Pointer	z = *(p ++);
		is equivalent to:
		z = *p;
		p = p + 1;

(*p)++	Post-Increment data pointed to by Pointer	z = (*p) ++;
		is equivalent to:
		z = *p;
		*p = *p + 1;

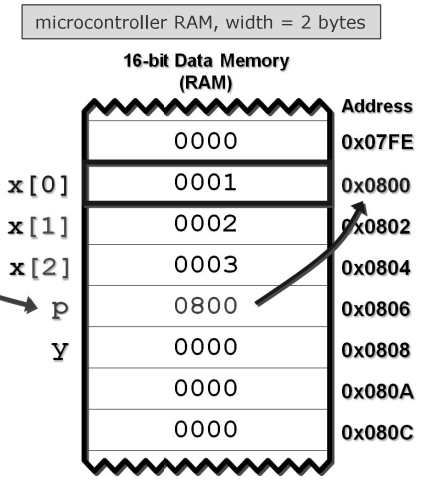
Remember: *(p++) is the same as *p++

electronics010.blogspot.com C language Eng. Mohamed Yousef

Post-Increment/Decrement Syntax Rule

Example

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(p++);

 y = 5 + (*p)++;
}
```

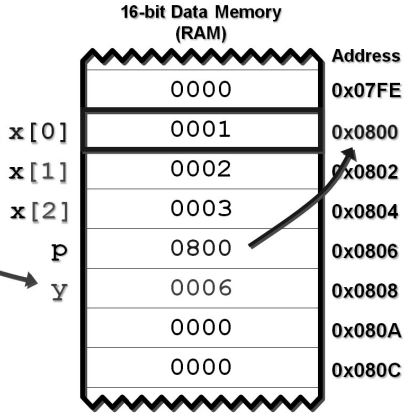


Post-Increment/Decrement Syntax Rule

Example

```
y = 5 + *p
y = 5 + x[0]
y = 5 + 1
y = 6
```

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(p++);

 y = 5 + (*p)++;
}
```

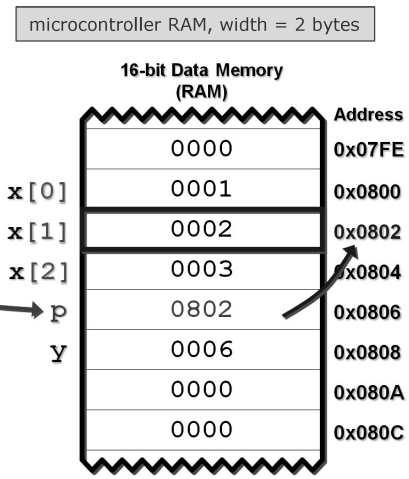


electronics010.blogspot.com C language Eng. Mohamed Yousef

Post-Increment/Decrement Syntax Rule

Example

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(p++);

 y = 5 + (*p)++;
}
```



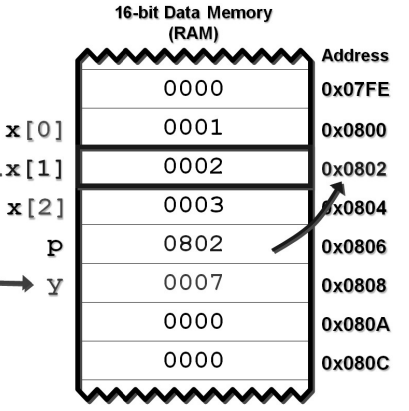
electronics010.blogspot.com C language Eng. Mohamed Yousef

Post-Increment/Decrement Syntax Rule

Example

```
y = 5 + *p
y = 5 + x[1]
y = 5 + 2
y = 7
```

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(p++);

 y = 5 + (*p)++;
}
```

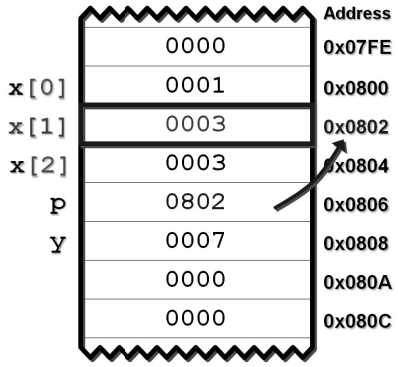


Post-Increment/Decrement Syntax Rule

Example

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(p++);

 y = 5 + (*p)++;
}
```

microcontroller RAM, width = 2 bytes



Pre-Increment/Decrement Syntax Rule

Syntax	Operation	Description by Example

*++p
*(++p)
	Pre-Increment Pointer	z = *(++ p);
		is equivalent to:
		p = p + 1;
		z = *p;

++(*p)	Pre-Increment data pointed to by Pointer	z = ++(*p);
		is equivalent to:
		*p = *p + 1;
		z = *p;

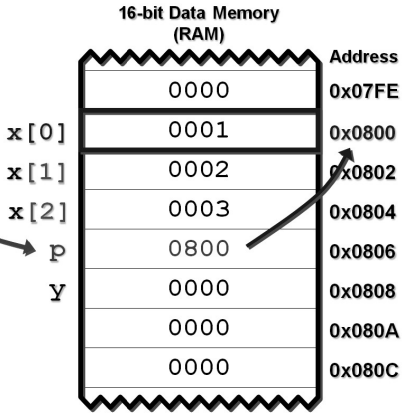
Remember: *(++p) is the same as *++p

Pre-Increment/Decrement Syntax Rule

Example

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(++p);

 y = 5 + ++(*p);
}
```

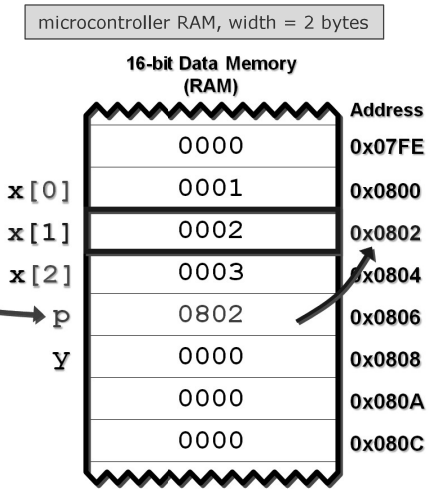
microcontroller RAM, width = 2 bytes



Pre-Increment/Decrement Syntax Rule

Example

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(++p);

 y = 5 + ++(*p);
}
```



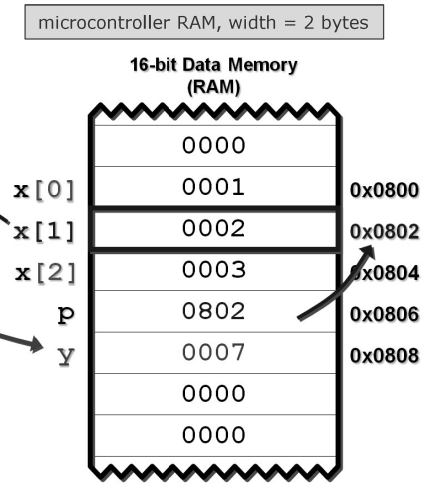
electronics010.blogspot.com C language Eng. Mohamed Yousef

Pre-Increment/Decrement Syntax Rule

Example

```
y = 5 + *p
y = 5 + x[1]
y = 5 + 2
y = 7
```

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(++p);

 y = 5 + ++(*p);
}
```

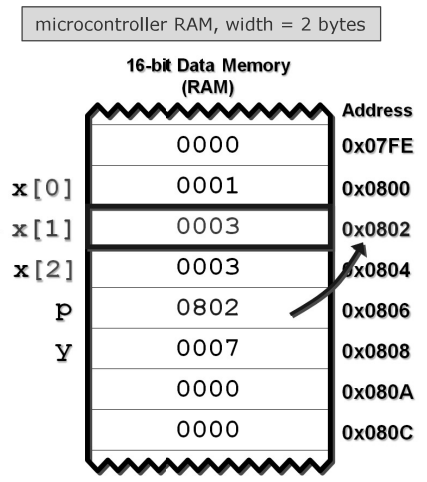


electronics010.blogspot.com C language Eng. Mohamed Yousef

Pre-Increment/Decrement Syntax Rule

Example

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(++p);

 y = 5 + ++(*p);
}
```



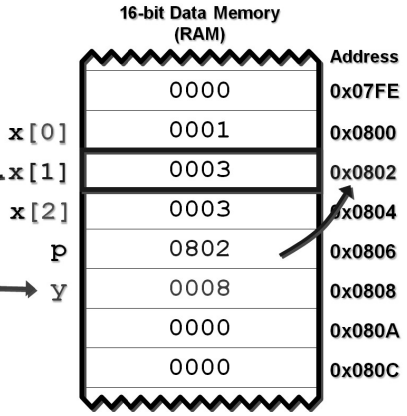
Pre-Increment/Decrement Syntax Rule

Example

```
y = 5 + *p
y = 5 + x[1]
y = 5 + 3
y = 8
```

```
{
 int x[3] = {1,2,3};
 int y;
 int *p = &x;

 y = 5 + *(++p);

 y = 5 + ++(*p);
}
```

microcontroller RAM, width = 2 bytes



electronics010.blogspot.com C language Eng. Mohamed Yousef

Pre-Increment/Decrement Syntax Rule

Summary

Modify the pointer itself

pre-inc: *(++p) or *++p or ++p
post-inc: *(p++) or *p++ or p++

Modify the value pointed to by the pointer

++(*p) and (*p)++

electronics010.blogspot.com C language Eng. Mohamed Yousef

Pre-Increment/Decrement Syntax Rule

Summary

Expression	*Address p*	*Value *p*
*p++	Incremented after the value is read	Unchanged
*(p++)	Incremented after the value is read	Unchanged
(*p)++	Unchanged	Incremented after it's read
*++p	Incremented before the value is read	Unchanged
*(++p)	Incremented before the value is read	Unchanged
++*p	Unchanged	Incremented before it's read
++(*p)	Unchanged	Incremented before it's read

Substituting pointers for array notation

Table 19-2	Array Notation Replaced by Pointers
***Array* alpha[]**	***Pointer* a**
alpha[0]	*a
alpha[1]	*(a+1)
alpha[2]	*(a+2)
alpha[3]	*(a+3)
alpha[*n*]	*(a+*n*)

Substituting pointers for array notation

```c
#include <stdio.h>

int main(){
int numbers[5] = {4,9,6,5,3};
int *ptr = numbers;

    printf("\n 3rd element = %d", numbers[2]);
    printf("\n 3rd element = %d", *(ptr + 2));

    return 0;
}
```

```
3rd element = 6
3rd element = 6
```


Substituting pointers for array notation

```c
#include <stdio.h>

int main(){
short a[8] = { 10, 20, 30, 40, 50, 60, 70, 80 };
short *p = a;

    printf("\n %i", a[3]);
    printf("\n %i", p[3]);

    printf("\n %i", *(a + 3));
    printf("\n %i", *(p + 3));

    return 0;
}
```

```
40
40
40
40
```

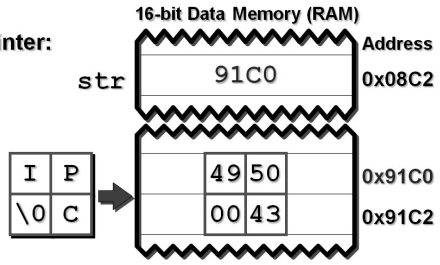
Pointers and Strings

- A string may be declared using a pointer just like it was with a char array, but now we use a pointer variable (no square brackets) instead of an array variable.
- The string may be initialized when it is declared, or it may be assigned later.
- The string itself will be stored in memory, and the pointer will be given the address of the first character of the string.

microcontroller RAM, width = 2 bytes

String declaration with a pointer:

```
char *str = "PIC";
```



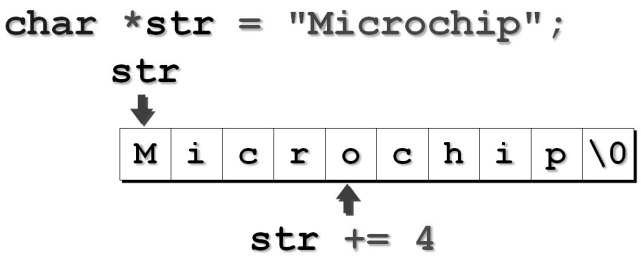
Please note that implementation varies depending on compiler and architecture used.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers and Strings

When initialized, a pointer to a string points to the first character:
You can then increment or add an offset to the pointer to access subsequent characters.

Example



electronics010.blogspot.com C language Eng. Mohamed Yousef

Pointers and Strings

Pointer versus Array

Initialization at Declaration

Initializing a character string when it is declared is essentially the same for both a pointer and an array:

Example: Pointer Variable

```
1  char *str = "PIC";
```

Example: Array Variable

```
1  char str[] = "PIC";
2
3  OR
4
5  char str[4] = "PIC";
```

The NULL character '\0' is automatically appended to strings in both cases (array must be large enough).

Pointers and Strings

Pointer versus Array

Assignment in Code

An entire string may be assigned to a **pointer** and a character **array** must be assigned character by character.

Example: Pointer Variable

```c
char *str;

str = "PIC";
```

Example: Array Variable

```c
char str[4];

str[0] = 'P';
str[1] = 'I';
str[2] = 'C';
str[3] = '\0';
```

Must explicitly add NULL character '\0' to array.

Printing Strings

```c
#include <stdio.h>

int main(){
char other[] = "Tony Blurt";
int x = 0;
    printf("\n ");

    while(other[x] != '\0'){
            printf("%c", other[x++]);
    }

    return 0;
}
```

```
Tony Blurt
```

```c
#include <stdio.h>

int main(){
char *other = "Tony Blurt";

    printf("\n ");

    while(*other != '\0'){
            printf("%c", *(other++));
    }

    return 0;
}
```

```
Tony Blurt
```


Printing Strings

```c
#include <stdio.h>

int main(){
char other[] = "Tony Blurt";

    printf("\n %s", &other[0]);

    return 0;
}
```

```
Tony Blurt
```

```c
#include <stdio.h>

int main(){
char other[] = "Tony Blurt";

    printf("\n %s", other);

    return 0;
}
```

```
Tony Blurt
```

Array is passed into any function as an address.

Printing Strings

```c
#include <stdio.h>

int main(){
char *other = "Tony Blurt";

    printf("\n %s", &other[0]);

    return 0;
}
```

```
Tony Blurt
```

```c
#include <stdio.h>

int main(){
char *other = "Tony Blurt";

    printf("\n %s", other);

    return 0;
}
```

```
Tony Blurt
```

Array is passed into any function as an address.

Printing Strings

```c
#include <stdio.h>

int main(){
char *other = "Tony Blurt";

    // print each memory address & its content
    while(*other != '\0'){

        printf("\n %c --> %p", *other, other);
        other++;

    }

    return 0;
}
```

```
T --> 00403024
o --> 00403025
n --> 00403026
y --> 00403027
  --> 00403028
B --> 00403029
l --> 0040302A
u --> 0040302B
r --> 0040302C
t --> 0040302D
```


Array Of Pointers

An array of pointers is an ordinary array variable whose elements happen to be all pointers.

Declaration

```c
char *p[4];
```

Initialization

```c
p[0] = &x;
```

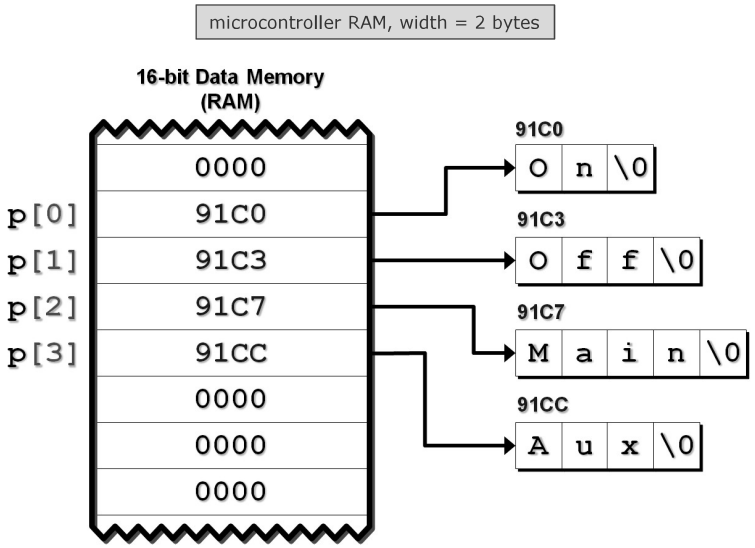
This example assigns the address of x to the pointer in element of the array.

```c
p[0] = "My string";
```

This example assigns the address of the first character in the string to the pointer in element 0 of the array.

Array Of Pointers

```
p[0] = "On";
p[1] = "Off";
p[2] = "Main";
p[3] = "Aux";
```



electronics010.blogspot.com C language Eng. Mohamed Yousef

Array Of Pointers

Dereferencing

To use the value pointed to by a pointer array element, just dereference it like you would an ordinary variable:

```c
y = *p[0];
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Array Of Pointers

```c
#include <stdio.h>

int main(){
int x;
char *fruit[] = {
    "watermelon",
    "banana",
    "pear",
    "apple",
    "coconut",
    "grape",
    "blueberry"
};

    for(x = 0; x < 7; x++){
        printf("\n %s", fruit[x]);
    }


    return 0;
}
```

```
watermelon
banana
pear
apple
coconut
grape
blueberry
```

Array Of Pointers

```
#include <stdio.h>

int main(){
int x = 0;
char *fruit[7] = {0};

    fruit[0] = "watermelon";
    fruit[1] = "banana";
    fruit[2] = "pear";
    fruit[3] = "apple";
    fruit[4] = "coconut";
    fruit[5] = "grape";
    fruit[6] = "blueberry";

    for(x = 0; x < 7; x++){
        printf("\n %s", fruit[x]);
    }

    return 0;
}
```

```
watermelon
banana
pear
apple
coconut
grape
blueberry
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Array Of Pointers

```
#include <stdio.h>
#define string char*

int main(){
string title = "Some of fruits:";
int x = 0;
string fruit[7] = {0};

    fruit[0] = "watermelon";
    fruit[1] = "banana";
    fruit[2] = "pear";
    fruit[3] = "apple";
    fruit[4] = "coconut";
    fruit[5] = "grape";
    fruit[6] = "blueberry";

    printf("\n %s", title);

    for(x = 0; x < 7; x++){
        printf("\n %s", fruit[x]);
    }

    return 0;
}
```

```
Some of fruits:
watermelon
banana
pear
apple
coconut
grape
blueberry
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Passing Pointers to Functions

Passing Parameters By Reference

```
#include <stdio.h>

void  By_Ref (int *x);

int main(){
    int a = 6;

    printf("\n a before calling = %d", a);
    /*
        &a indicates pointer to a ie. address of variable a
    */
    By_Ref(&a);
    printf("\n a after calling = %d", a);

    return 0;
}

//function definition
void  By_Ref (int *x){

    *x = 10;
}
```

```
a before calling = 6
a after calling = 10
```

Passing Pointers to Functions — Passing Parameters By Reference

A function with a pointer parameter, for example `int foo(int *q)` must be called in one of two ways:
(assume: `int x`, `*p = &x;`)

`foo( &x )`	Pass an address to the function so the address may be assigned to the pointer parameter: `q = &x`
`foo ( p )`	Pass a pointer to the function so the address may be assigned to the pointer parameter: `q = p`

electronics010.blogspot.com C language Eng. Mohamed Yousef

Passing Pointers to Functions — Passing Parameters By Reference

```c
#include <stdio.h>
void Swap (int *, int *);

int main(){
int x = 5, y = 10;
int *p = &x;

    Swap(p, &y);
    printf("\n x = %d : y = %d", x, y);

    return 0;
}

void Swap (int *n1, int *n2){
int  temp = 0;

    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

```
x = 10 : y = 5
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Passing Pointers to Functions

```c
#include <stdio.h>
int StingLength(char []);

int main(){
char text[] = "Embedded Systems";

    printf("\n Length =  %d characters", StingLength(text));

    return 0;
}

int StingLength(char string[]){
int x = 0;

    while( string[x] != '\0' ){
        x++;
    }

    return x;
}
```

```
Length =  16 characters
```

Array exhibits a special behaviour. Whenever you refer an array name directly, is behaves a pointer pointing at zeroth array element.
Which means both of the below statements are equivalent.

Passing Pointers to Functions

```
#include <stdio.h>
int StingLength(char *);

int main(){
char *text = "Embedded Systems";

    printf("\n Length =  %d characters", StingLength(text));

    return 0;
}

int StingLength(char *string){
int x = 0;

    while( *string != '\0' ){
        x++;
        string++;
    }

    return x;
}
```

```
Length =  16 characters
```


THE CONST QUALIFIER

Passing arguments by reference, provides C programmers the capability of modifying argument contents via pointers.

You can also use the *const* qualifier in conjunction with pointers to achieve a read-only argument while still achieving the pass by reference capability.

THE CONST QUALIFIER

```
#include <stdio.h>
void printArray(const int []);

int main(){
int numbers[3] = {2, 4, 6};

    printArray(numbers);

    return 0;
}

void printArray(const int num[]){
int x = 0;

    printf("\n Array contents are: ");

    for (x = 0; x < 3; x++){
        printf("%d ", num[x]);
    }
}
```

```
Array contents are: 2 4 6
```

THE CONST QUALIFIER

```
#include <stdio.h>
void modifyArray(const int []);

int main(){
int numbers[3] = {2, 4, 6};

    modifyArray(numbers);

    return 0;
}

void modifyArray(const int num[]){
int x = 0;

    for (x = 0; x < 3; x++){
        num[x] = num[x] * num[x]; //this will not work!
    }
}
```

X
ERROR

```
error: assignment of read-only location '*(num + (sizetype)((unsigned int)x...
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```


Returning a pointer from a function

You can declare *pointer functions*, which return a memory location as a value.

- The value that's returned must be declared as a static variable.
- Keep in mind that variables are local to their functions.
- You must retain the data in the variable by declaring it as a static type so that its contents aren't discarded when the function stops.

Returning a pointer from a function

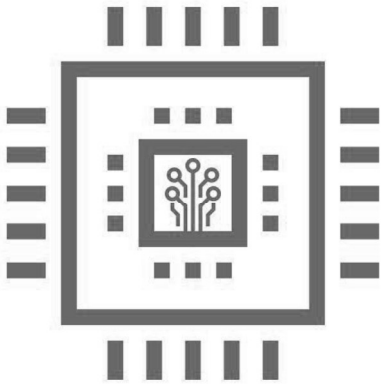
```
/*
    Reversing a String
*/
#include <stdio.h>

char *strrev(char *);

int main(){
char *string = "E N G I N E E R";

    printf("\n %s", strrev(string));

    return 0;
}

char *strrev(char *input){

static char output[20];

char *i = input;
char *o = output;

    /* get address of last char.
       and store it in pointer i
    */
    while(*i != '\0'){
        /* increment address till
           end of 'input'
        */
        i++;
    }
    i--;

    while(i >= input){
        *o = *i;
        i--;
        o++;
    }

    // end of string
    *o = '\0';

    return(output);
}
```

```
R E E N I G N E
```

LAB

Embedded Systems

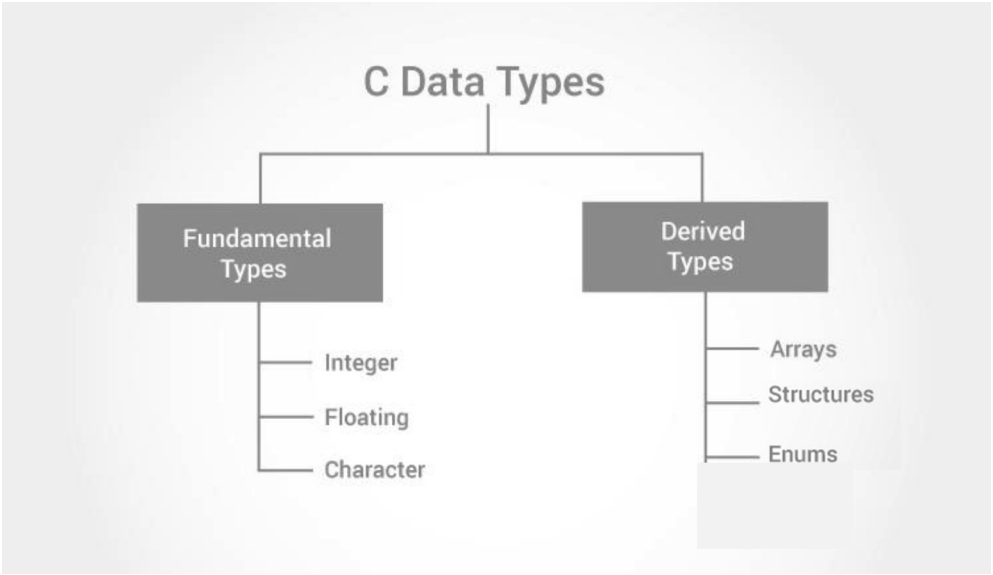
C language



15- Structures

Eng. Mohamed Yousef

Types



electronics010.blogspot.com C language Eng. Mohamed Yousef

Structures

Structure is a collection of variables of different types under a single name.

name	age	salary
Ali	35	5000
Huda	29	4000

Structure Definition

structure tag

```
struct employee{
    char name[15];
    char age;
    int salary;
};
```

structure members

name	age	salary
Ali	35	5000
Huda	29	4000

electronics010.blogspot.com C language Eng. Mohamed Yousef

Structure Definition

```
struct employee{
    char name[15];
    char age;
    int salary;
};
```

name	age	salary
Ali	35	5000
Huda	29	4000

Points to remember while structure definition

- You must terminate structure definition with semicolon `;` .
- You cannot assign value to members inside the structure definition, it will cause compilation error.
- You can define a structure anywhere like global scope (accessible by all functions) or local scope (accessible by particular function).
- Structure member definition may contain other structure type.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Structure Definition

⚠ C structures cannot have static members

```
// C program with structure static member
struct Record {
    static int x;
};

// Driver program
int main()
{
    return 0;
}
```

X
ERROR

Declaring a Structure Variable

```
struct employee{
    char name[15];
    char age;
    int salary;
};
```

name	age	salary
Ali	35	5000
Huda	29	4000

In C programming, there are two ways to declare a structure variable:

1. Along with structure definition
2. After structure definition

electronics010.blogspot.com C language Eng. Mohamed Yousef

Declaring a Structure Variable

```
struct employee{
    char name[15];
    char age;
    int salary;
} emp1;
```

structure variable

name	age	salary
Ali	35	5000
Huda	29	4000

```
struct employee{
    char name[15];
    char age;
    int salary;
} emp1, emp2;
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Declaring a Structure Variable

```
struct employee{
    char name[15];
    char age;
    int salary;
};

struct employee emp1;
```

structure type structure variable

name	age	salary
Ali	35	5000
Huda	29	4000

```
struct employee{
    char name[15];
    char age;
    int salary;
};

struct employee emp1, emp2;
```

initialize structure members

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
   int x = 0;  // COMPILER ERROR:
   int y = 0;  // COMPILER ERROR:
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

electronics010.blogspot.com C language Eng. Mohamed Yousef

initialize structure members

```
struct employee{
    char name[15];
    char age;
    int salary;
};

/* Structure members can be
initialized using curly braces {}.
*/
struct employee emp1 = {{'0'},0 ,0};
```

name	age	salary
Ali	35	5000
Huda	29	4000

Note: Since your first member in the structure is an array:
Outer braces are for the struct, inner braces are for the array.

electronics010.blogspot.com C language Eng. Mohamed Yousef

initialize structure members

```
struct employee{
    char name[15];
    char age;
    int salary;
};

struct employee emp1 = {"Ali", 35, 5000};
```

name	age	salary
Ali	35	5000
Huda	29	4000

Accessing Structure Members

C supports two operators to access structure members, using a structure variable.

- Dot/period operator `.`
- Arrow operator `->`

name	age	salary
Ali	35	5000
Huda	29	4000

electronics010.blogspot.com C language Eng. Mohamed Yousef

Accessing Structure Members

Dot/period operator (.) in C

Dot/period operator also known as member access operator.

name	age	salary
Ali	35	5000
Huda	29	4000

```c
#include <stdio.h>

struct employee{
    char name[15];
    char age;
    int salary;
};

struct employee emp1 = {"Ali", 35, 5000};

int main(){

    printf("\n Name:  %s", emp1.name);
    printf("\n Age:   %d", emp1.age);
    printf("\n Salary %d", emp1.salary);

    return 0;
}
```

```
Name:  Ali
Age:   35
Salary 5000
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Accessing Structure Members

name	age	salary
Ali	35	5000
Huda	29	4000

```c
#include <stdio.h>

struct employee{
    char name[15];
    char age;
    int salary;
};

// Declaring a Structure Variable
struct employee emp1;

int main(){
    //assign values to members
    emp1.name[0] = 'A';
    emp1.name[1] = 'l';
    emp1.name[2] = 'i';
    emp1.name[3] = '\0';

    emp1.age = 35;
    emp1.salary = 5000;

    printf("\n Name:  %s", emp1.name);
    printf("\n Age:   %d", emp1.age);
    printf("\n Salary %d", emp1.salary);

    return 0;
}
```

```
Name:  Ali
Age:   35
Salary 5000
```

Accessing Structure Members

```c
#include <stdio.h>

struct employee{
    char *name;
    char age;
    int salary;
};

// Declaring a Structure Variable
struct employee emp1;

int main(){
    //assign values to members
    emp1.name = "Ali" ;
    emp1.age = 35;
    emp1.salary = 5000;

    printf("\n Name:  %s", emp1.name);
    printf("\n Age:   %d", emp1.age);
    printf("\n Salary %d", emp1.salary);

    return 0;
}
```

name	age	salary
Ali	35	5000
Huda	29	4000

```
Name:  Ali
Age:   35
Salary 5000
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Keyword typedef

The *typedef* keyword is used for creating structure definitions to build an alias relationship with the structure tag (structure name).

It provides a shortcut for programmers when creating instances of the structure.

```c
typedef struct employee {  //modification here
    char *name;
    char age;
    int salary;
} Emp;    //modification here

Emp emp1;    //modification here
```

```c
typedef struct {  //modification here
    char *name;
    char age;
    int salary;
} Emp;    //modification here

Emp emp1;    //modification here
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Nesting Structures

```c
#include <stdio.h>

typedef struct {
    int basic;
    int bonus;

}Salary;

typedef struct {
    char *name;
    char age;
    Salary salary;
} Emp;

Emp emp1;

int main(){
    //assign values to members
    emp1.name = "Ali" ;
    emp1.age = 35;
    emp1.salary.basic = 4000;
    emp1.salary.bonus = 1000;

    printf("\n Name:  %s", emp1.name);
    printf("\n Age:   %d", emp1.age);
    printf("\n Basic Salary: %d", emp1.salary.basic);
    printf("\n Bonus Salary: %d", emp1.salary.bonus);

    return 0;
}
```

name	age	salary
Ali	35	5000
Huda	29	4000

```
Name:  Ali
Age:   35
Basic Salary: 4000
Bonus Salary: 1000
```

Arrays of Structures

Like other primitive data types, we can create an array of structures.

```
typedef struct {
    char *name;
    char age;
    int salary;
} Emp;

Emp emp[2];
```


Initializing Arrays of Structures at Declaration

```
typedef struct {
    char *name;
    char age;
    int salary;
} Emp;

Emp emp[2] = {
    {"Ali", 35, 5000},
    {"Huda", 29, 4000},
};
```

name	age	salary
Ali	35	5000
Huda	29	4000

Using Arrays of Structures

```
#include <stdio.h>

typedef struct {
    char *name;
    char age;
    int salary;
} Emp;

int main(){
Emp emp[2];
int x = 0;

    emp[0].name = "Ali" ;
    emp[0].age = 35;
    emp[0].salary = 5000;

    emp[1].name = "Huda" ;
    emp[1].age = 29;
    emp[1].salary = 4000;

    for(x = 0; x < 2; x++){
        printf("\n Name:  %s", emp[x].name);
        printf("\n Age:   %d", emp[x].age);
        printf("\n Basic Salary: %d", emp[x].salary);
        printf("\n -------------------");
    }

    return 0;
}
```

name	age	salary
Ali	35	5000
Huda	29	4000

```
Name:  Ali
Age:   35
Basic Salary: 5000
-----------------
Name:  Huda
Age:   29
Basic Salary: 4000
-----------------
```

Passing structures to a function

There are mainly two ways to pass structures to a function:

1. Passing by value
2. Passing by reference

Passing structures to a function

```c
#include <stdio.h>

typedef struct {
    char *name;
    char age;
    int salary;
} Emp;
// ---------------------------------------------
void By_Value(Emp);
// ---------------------------------------------
int main(){
Emp emp = {"Ali", 35, 5000};

    By_Value(emp);

    printf("\n Name:  %s", emp.name);
    printf("\n Age:   %d", emp.age);
    printf("\n Basic Salary: %d", emp.salary);
    printf("\n ------------------");

    return 0;
}
// ---------------------------------------------
```

```c
void By_Value(Emp emp){

    emp.name = "Huda" ;
    emp.age = 29;
    emp.salary = 4000;

    printf("\n Name:  %s", emp.name);
    printf("\n Age:   %d", emp.age);
    printf("\n Basic Salary: %d", emp.salary);
    printf("\n ------------------");
}
// ---------------------------------------------
```

```
Name:  Huda
Age:   29
Basic Salary: 4000
------------------
Name:  Ali
Age:   35
Basic Salary: 5000
------------------
```


structure pointer

Like primitive types, we can have pointer to a structure.

If we have a pointer to structure, members are accessed using arrow (->) operator (structure pointer operator) .

structure pointer

```c
#include <stdio.h>

typedef struct {
    char *name;
    char age;
    int salary;
} Emp;

int main(){
Emp emp = {"Ali", 35, 5000};
Emp *ptr_emp = &emp;

    printf("\n Name:  %s", ptr_emp->name);
    printf("\n Age:   %d", ptr_emp->age);
    printf("\n Basic Salary: %d", ptr_emp->salary);
    printf("\n -------------------");

    return 0;
}
```

```
Name:  Ali
Age:   35
Basic Salary: 5000
-----------------
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Passing structures to a function

```c
#include <stdio.h>

typedef struct {
    char *name;
    char age;
    int salary;
} Emp;
// ---------------------------------------------
void By_Ref(Emp *);
// ---------------------------------------------
int main(){
Emp emp = {"Ali", 35, 5000};
Emp *ptr_emp = &emp;

    By_Ref(ptr_emp);

    printf("\n Name:  %s", ptr_emp->name);
    printf("\n Age:   %d", ptr_emp->age);
    printf("\n Basic Salary: %d", ptr_emp->salary);
    printf("\n -------------------");

    return 0;
}
// ---------------------------------------------
```

```c
void By_Ref(Emp *ptr_emp){

    ptr_emp->name = "Huda" ;
    ptr_emp->age = 29;
    ptr_emp->salary = 4000;

    printf("\n Name:  %s", ptr_emp->name);
    printf("\n Age:   %d", ptr_emp->age);
    printf("\n Basic Salary: %d", ptr_emp->salary);
    printf("\n -------------------");
}
// ---------------------------------------------
```

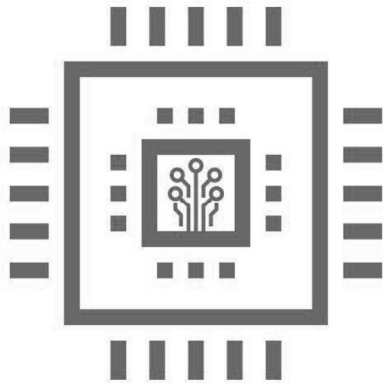
```
Name:  Huda
Age:   29
Basic Salary: 4000
-----------------
Name:  Huda
Age:   29
Basic Salary: 4000
-----------------
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

Embedded Systems

C language



16- Unions

Eng. Mohamed Yousef

Unions

- *Unions* are quite similar to structures.
- Defining a *union* is as easy as replacing the keyword *struct* with the keyword union.
- The member of unions can be accessed in similar manner as structures.

Difference between union and structure

Whereas structures reserve separate memory segments for each member when they are created, a *union reserves a single memory space for its largest member*, thereby providing a memory-saving feature for members to share the same memory space.

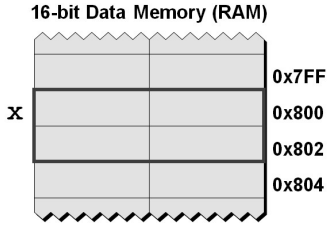
electronics010.blogspot.com C language Eng. Mohamed Yousef

Unions

microcontroller RAM, width = 2 bytes

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

Space allocated for `x` is size of `float`

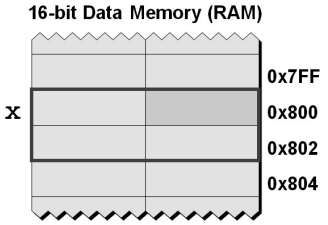


Unions

microcontroller RAM, width = 2 bytes

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

x.a only occupies the lowest byte of the union



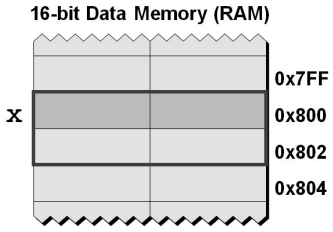
electronics010.blogspot.com C language Eng. Mohamed Yousef

Unions

microcontroller RAM, width = 2 bytes

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

x.b only occupies the lowest two bytes of the union



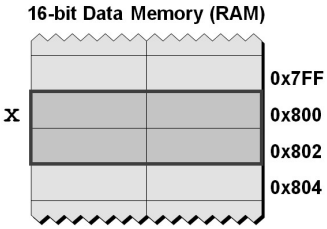
electronics010.blogspot.com C language Eng. Mohamed Yousef

Unions

microcontroller RAM, width = 2 bytes

```
typedef union
{
  char a;
  int b;
  float c;
} mixedBag;

mixedBag x;
```

x.c occupies all four bytes of the union



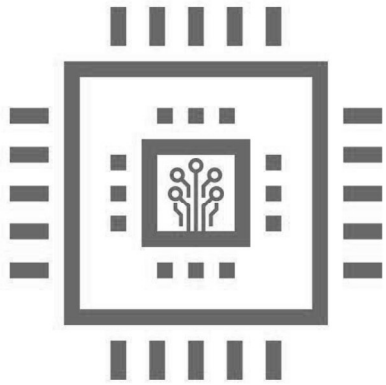
Unions

```c
#include <stdio.h>

typedef union {
    unsigned char x;
    unsigned int y;
} Num;

int main(){
Num num;

    num.x = 'A';
    printf("\n x: %c", num.x);

    num.y = 54321;
    printf("\n y: %d", num.y);
    printf("\n ---------------");

    num.x = 'A';
    num.y = 54321;

    printf("\n x: %c", num.x);
    printf("\n y: %d", num.y);
    printf("\n ---------------");

    num.y = 54321;
    num.x = 'A';

    printf("\n x: %c", num.x);
    printf("\n y: %d", num.y);
    printf("\n ---------------");

    return 0;
}
```

```
x: A
y: 54321
---------------
x: 1
y: 54321
---------------
x: A
y: 54337
---------------
```

Embedded Systems

C language



18- Enumeration

Eng. Mohamed Yousef

Enumeration

Enumeration (or *enum*) is a user defined data type in C.

It is mainly used to assign names to integral constants.

Values in an enum start with 0, unless specified otherwise, and are incremented by 1.

```
enum months {
   JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // end enum months
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Enumeration

To number the months 1 to 12, use the following enumeration:

```
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // end enum months
```

Enumeration

```
enum State {Working = 1, Failed = 0};
```

```
1   enum people {Rob, Steve, Paul = 7, Bill, Gary};
```

```
Label Values:
Rob = 0, Steve = 1, Paul = 7, Bill = 8, Gary = 9
```


Enumeration

```
#include <stdio.h>

enum State {Working = 1, Failed = 0, Freezed = 0};

int main(){

    printf("\n %d, %d, %d", Working, Failed, Freezed);

    return 0;
}
```

```
1, 0, 0
```


Enumeration

```
#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,
          Aug, Sep, Oct, Nov, Dec};

int main()
{
   int i;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);

   return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Enumeration

Variables of type enum can also be defined. They can be defined in two ways:

```
// In both of the below cases, "day" is
// defined as the variable of type week.

enum week{Mon, Tue, Wed};
enum week day;

// Or

enum week{Mon, Tue, Wed}day;
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Enumeration

```
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Output:

```
2
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Enumeration

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Output:

```
The day number stored in d is 4
```

Enumeration

```c
#include <stdio.h>

// initialize array of pointers
const char *monthName[13] = { "", "January", "February", "March",
                              "April", "May", "June", "July", "August",
                              "September", "October", "November",
                              "December" };

// enumeration constants represent months of the year
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC};

int main(){
enum months month; // can contain any of the 12 months

    // loop through months
    for (month = JAN; month <= DEC; month++ ){

        printf( "\n %2d  %s", month, monthName[month] );
    }

    return 0;
}
```

```
 1   January
 2   February
 3   March
 4   April
 5   May
 6   June
 7   July
 8   August
 9   September
10   October
11   November
12   December
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Enumeration

```c
#include <stdio.h>

// initialize array of pointers
const char *monthName[13] = { "", "January", "February", "March",
                              "April", "May", "June", "July", "August",
                              "September", "October", "November",
                              "December" };

// enumeration constants represent months of the year
typedef enum {JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC} months;

int main(){
months month; // can contain any of the 12 months

    // loop through months
    for (month = JAN; month <= DEC; month++ ){

        printf( "\n %2d  %s", month, monthName[month] );
    }

    return 0;
}
```

```
 1   January
 2   February
 3   March
 4   April
 5   May
 6   June
 7   July
 8   August
 9   September
10   October
11   November
12   December
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Enumeration

```c
#include <stdio.h>

typedef enum {
    false,
    true} boolean;

int main(){
boolean error = false;

    if(error){
        printf("\n There is an error!");
    } else {
        printf("\n No error occurred.");
    }

    return 0;
}
```

```
No error occurred.
```

Enumeration

Enum vs Macro

We can also use macros to define names constants. For example we can define 'Working' and 'Failed' using following macro.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.

a) Enums follow scope rules.

b) Enum variables are automatically assigned values. Following is simpler

```
enum state  {Working, Failed, Freezed};
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

electronics010.blogspot.com C language Eng. Mohamed Yousef

Exercise:

Predict the output of following C programs

Program 1:

```
#include <stdio.h>
enum day {sunday = 1, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Exercise:

Predict the output of following C programs

Program 1:

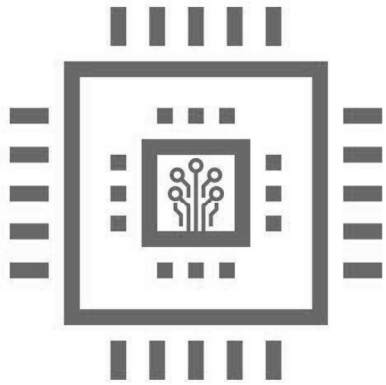
```
#include <stdio.h>
enum day {sunday = 1, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Output:

```
The day number stored in d is 4
```

Embedded Systems

C language



17- Bit Fields

Eng. Mohamed Yousef

sizeof() operator

sizeof() is used to calculate the size (in bytes) of any data type in C.

Usage

sizeof() operator is used in different way according to the operand type.

1) When operand is a Data Type (int, float, char...) it simply returns amount of memory is allocated to that data types.

Note: *sizeof()* may give different output according to machine.

```c
#include <stdio.h>

int main(){
    printf(" char   = %d\n", sizeof(char));
    printf(" int    = %d\n", sizeof(int));
    printf(" float  = %d\n", sizeof(float));
    printf(" double = %d\n", sizeof(double));

    return 0;
}
```

```
char   = 1
int    = 4
float  = 4
double = 8
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

sizeof() operator

2) When operand is an expression. It returns size of the expression.

```c
#include <stdio.h>

int main(){
    int a = 0;
    double d = 10.21;
    printf(" size = %d", sizeof(a + d));

    return 0;
}
```

```
size = 8
```

sizeof() operator

Need of Sizeof.

1) Calculating the size of *struct* and *array*.

```
#include <stdio.h>

int main(){
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    printf("\n Array size :%d", sizeof(arr));

    return 0;
}
```

```
Array size :32
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

sizeof() operator

Need of Sizeof.

2) To find out number of elements in a array.

```
#include <stdio.h>

int main(){
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    printf("\n Number of elements :%d", sizeof(arr)/sizeof(arr[0]));

    return 0;
}
```

```
Number of elements :8
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

sizeof() operator

Need of Sizeof.

3) To allocate block of memory dynamically.

sizeof is greatly used in dynamic memory allocation. For example, if we want to allocate memory for which is sufficient to hold 10 integers and we don't know the sizeof(int) in that particular machine. We can allocate with the help of sizeof.

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

int main(){

    int *ptr = malloc(10*sizeof(int));

    return 0;
}
```

Bit Fields

- In C, we can specify size (in bits) of structure and union members.
- *Bit Fields* are ordinary members of a structure and have a specific bit width.
- The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bit Fields

Syntax

```
struct  structName
{
   unsigned int memberName1: bitWidth;
   ...
   unsigned int memberNamen:bitWidth
}
```

- Bit fields must be declared as unsigned int or int.
- The maximum value that can store in unsigned int filed is (2 power n) – 1
- and in int filed is 2 power (n – 1)
- Bit fields occupy a specified number of adjacent bits from one to *sizeof(int)*.

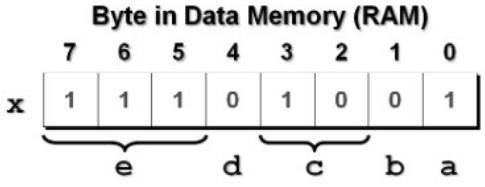
Example

```
1  typedef struct
2  {
3    unsigned int bit0: 1;
4    unsigned int bit1to3: 3;
5    unsigned int bit4: 1;
6    unsigned int bit5: 1;
7    unsigned int bit6to7: 2;
8  } byteBits;
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bit Fields

```
1  struct byteBits
2  {
3    unsigned a: 1;
4    unsigned b: 1;
5    unsigned c: 2;
6    unsigned d: 1;
7    unsigned e: 3;
8  } x;
9
10 int main(void)
11 {
12   x.a = 1;          //x.a may contain values from 0 to 1
13   x.b = 0;          //x.b may contain values from 0 to 1
14   x.c = 0b10;       //x.c may contain values from 0 to 3
15   x.d = 0x0;        //x.d may contain values from 0 to 1
16   x.e = 7;          //x.e may contain values from 0 to 7
17 }
```



Bit Fields

```c
#include <stdio.h>

// A simple representation of date
typedef struct {
   unsigned int d;
   unsigned int m;
   unsigned int y;
}Date;
// ------------------------------------------
int main(){
Date dt = {31, 12, 2014};

    printf("\n Struction size : %d bytes", sizeof(Date));
    printf("\n Date is %d/%d/%d", dt.d, dt.m, dt.y);

    return 0;
}
```

```
Struction size : 12 bytes
Date is 31/12/2014
```


Bit Fields

```c
#include <stdio.h>

// A space optimized representation of date
typedef struct {
   // d has value between 1 and 31, so 5 bits
   // are sufficient
   unsigned int d: 5;

   // m has value between 1 and 12, so 4 bits
   // are sufficient
   unsigned int m: 4;
   unsigned int y;
}Date;
// ------------------------------------------
int main(){
Date dt = {31, 12, 2014};

    printf("\n Struction size : %d bytes", sizeof(Date));
    printf("\n Date is %d/%d/%d", dt.d, dt.m, dt.y);

    return 0;
}
```

```
Struction size : 8 bytes
Date is 31/12/2014
```


Bit Fields

We cannot have pointers to bit field members

```c
#include <stdio.h>

// A space optimized representation of date
typedef struct {
   unsigned int d: 5;
   unsigned int m: 4;
   unsigned int y;
}Date;
// ------------------------------------------
int main(){
Date dt = {31, 12, 2014};

    printf("\n Address of dt.d is %p", &dt.d);

    return 0;
}
```


```
error: cannot take address of bit-field 'd'
```

Bit Fields

Array of bit fields is not allowed.

```
#include <stdio.h>

// A space optimized representation of date
typedef struct {
   unsigned int d[3]: 5;
   unsigned int m: 4;
   unsigned int y;
}Date;
// ----------------------------------------
int main(){
Date dt = {31, 12, 2014};

    printf("\n Struction size : %d bytes", sizeof(Date));

    return 0;
}
```

X
ERROR

```
error: bit-field 'd' has invalid type
```


LAB

Bit Fields

Exercise:

Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

1)

```
#include <stdio.h>
struct test
{
   unsigned int x;
   unsigned int y: 33;
   unsigned int z;
};
int main()
{
   printf("%d", sizeof(struct test));
   return 0;
}
```

Bit Fields

Exercise:

Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

1)

```
error: width of 'y' exceeds its type
```

```
#include <stdio.h>
struct test
{
   unsigned int x;
   unsigned int y: 33;
   unsigned int z;
};
int main()
{
   printf("%d", sizeof(struct test));
   return 0;
}
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bit Fields

Exercise:

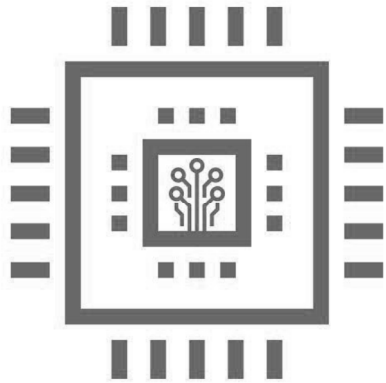
Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

3)

```
union test
{
  unsigned int x: 3;
  unsigned int y: 3;
  int z;
};

int main()
{
   union test t;
   t.x = 5;
   t.y = 4;
   t.z = 1;
   printf("t.x = %d, t.y = %d, t.z = %d",
           t.x, t.y, t.z);
   return 0;
}
```

Embedded Systems

C language



20- Memory Management

Eng. Mohamed Yousef

Memory Management

- In programming each variable, constant occupies space in memory, defined by their data type.
- The compiler reserves required memory (bytes) during compilation according to the specified data type.
- So you must exactly know how many bytes you require
- Many text also refer compile time memory allocation as static or stack memory allocation.
- The biggest disadvantage of compile time memory allocation, we do not have control on allocated memory.
- You cannot increase, decrease or free memory, the compiler take care of memory management.

Memory Management

For example

If you need to declare a variable to store marks of N students.

```
int marks[100];
```

Above declaration will occupy memory for 100 students (reserved memory bytes will be `100 * sizeof(int)`).

There are few issues with above declaration:

- Memory wastage:

If you are providing input of 10 students, then 90 * sizeof(int) memory gets wasted and cannot be used for other purposes.

- Less flexible:

You cannot alter memory size once allocated.

- No control:

You cannot clear allocated bytes from memory if you don't require at any stage.

Memory Management

Runtime or dynamic memory allocation

Memory allocated through malloc(), calloc(), realloc() or free() is called as runtime memory allocation.

You can also refer runtime memory allocation as dynamic or heap memory allocation.

To use those functions, you must include *stdlib.h* header file.

malloc() function

malloc() allocates N bytes in memory and return pointer to allocated memory.

Using that pointer you can access the memory allocated.

Syntax

```
void * malloc(number_of_bytes);
```

It returns *void* pointer (generic pointer). Which means we can easily typecast it to any other pointer types.

It accepts an integer number_of_bytes, i.e. total bytes to allocate in memory.

Note: *malloc()* returns NULL pointer on failure.

malloc() function

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

int main(){
char *name;

    name = (char *) malloc(80 * sizeof(char));

    if ( name == NULL ){
        printf("\n Out of memory!\n");
    } else {
        printf("\n Memory allocated.\n");
    }

    return 0;
}
```

```
Memory allocated.
```

free() function

The *free()* function clears the dynamically allocated memory.

If pointer contains NULL, then free() does nothing

Syntax

```
free(ptr);
```


malloc() function

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

int main(){
char *name;

    name = (char *) malloc(80 * sizeof(char));

    if ( name == NULL ){
        printf("\n Out of memory!\n");
        return 0;
    }

    printf("\n Memory allocated.\n");

    free(name);

    return 0;
}
```

```
Memory allocated.
```


calloc() function

calloc() function allocates memory contiguously and initializes all blocks with 0.

Syntax

```
void* calloc(number_of_blocks, number_of_bytes);
```

`number_of_blocks` → total blocks to allocate

`number_of_bytes` → bytes to allocate per block.

Note: It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc.

calloc() function

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

int main(){
char *name;

    name = (char *) calloc(80, sizeof(char));

    if ( name == NULL ){
        printf("\n Out of memory!\n");
        return 0;
    }

    printf("\n Memory allocated.\n");

    free(name);

    return 0;
}
```

```
Memory allocated.
```


Working with Memory Segments

Individual memory segments acquired by malloc() can be treated much like array members; these memory segments can be referenced with indexes.

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

int main(){
int *numbers;
int x;

    numbers = (int *) malloc(5 * sizeof(int));
    if ( numbers == NULL ){
        return 0; // return if malloc is not successful
    }

    numbers[0] = 100;
    numbers[1] = 200;
    numbers[2] = 300;
    numbers[3] = 400;
    numbers[4] = 500;

    printf("\n Individual memory segments initialized to:\n");
    for ( x = 0; x < 5; x++ ){
        printf("\n numbers[%d] = %d", x, numbers[x]);
    }

    return 0;
}
```

```
Individual memory segments initialized to:

numbers[0] = 100
numbers[1] = 200
numbers[2] = 300
numbers[3] = 400
numbers[4] = 500
```


Working with Memory Segments

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

int main(){
int *numbers;
int x;

    numbers = (int *) malloc(5 * sizeof(int));
    if ( numbers == NULL ){
        return 0; // return if malloc is not successful
    }

    *numbers       = 100;
    *(numbers + 1) = 200;
    *(numbers + 2) = 300;
    *(numbers + 3) = 400;
    *(numbers + 4) = 500;

    printf("\n Individual memory segments initialized to:\n");
    for ( x = 0; x < 5; x++ ){
        printf("\n number %d = %d", x + 1, *(numbers + x));
    }

    return 0;
}
```

```
Individual memory segments initialized to:

number 1 = 100
number 2 = 200
number 3 = 300
number 4 = 400
number 5 = 500
```

realloc() function

We use *realloc()* function to update the size of exiting allocated memory blocks.
The function may resize or move the allocated memory blocks to a new location.

Syntax

```
void* realloc(ptr, updated_memory_size);
```

`ptr` → pointer to previously allocated memory.
`updated_memory_size` → new (existing + new) size of the memory block.

There are three scenarios for realloc()'s outcome :

Scenario	Outcome
Successful without move	Same pointer returned
Successful with move	New pointer returned
Not successful	NULL pointer returned

electronics010.blogspot.com C language Eng. Mohamed Yousef

realloc() function

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

int main(){
int *numbers, *new_numbers;
int x;

    numbers = (int *) malloc(5 * sizeof(int));
    if ( numbers == NULL ){
        printf("\n Out of memory!");
        return 0;
    }

    for(x = 0; x < 5; ++x){
        *(numbers + x) = 100 * (x + 1);
    }

    new_numbers = (int *) realloc(numbers, 8 * sizeof(int));
    if ( new_numbers == NULL ){
        printf("\n Out of memory!");
        return 0;
    }
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

realloc() function

```c
    for (x = 5; x < 8; ++x){
        *(new_numbers + x) = 100 * (x + 1);
    }

    for (x = 0; x < 8; ++x){
        printf("\n Number %d = %d", x + 1, *(new_numbers + x));
    }

    free(new_numbers);
    return 0;
}
```

```
Number 1 = 100
Number 2 = 200
Number 3 = 300
Number 4 = 400
Number 5 = 500
Number 6 = 600
Number 7 = 700
Number 8 = 800
```

LAB

electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
/*
 * C program to demonstrate ralloc() and free() function.
 */
#include <stdio.h>
#include <stdlib.h>

int main(){
int x, max, new_size;;
int *ptr_numbers;

    // Input maximum elements of array
    printf("\n Enter total number of elements: ");
    scanf("%d", &max);

    // Allocate memory for 'max' integer elements
    ptr_numbers = (int *) malloc(max * sizeof(int));
    if ( ptr_numbers == NULL ){
        printf("\n Out of memory!");
        return 0;
    }

    // Input elements from user
    printf("\n Enter %d elements: \n", max);

    for(x = 0; x < max; ++x){
        scanf("%d", (ptr_numbers + x));
    }
```

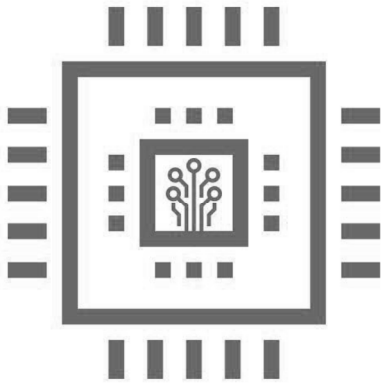
electronics010.blogspot.com C language Eng. Mohamed Yousef

```c
    // Reallocate memory
    printf("\n Enter new size of the array: ");
    scanf("%d", &new_size);

    ptr_numbers = (int *) realloc(ptr_numbers, new_size * sizeof(int));
    if ( ptr_numbers == NULL ){
        printf("\n Out of memory!");
        return 0;
    }

    // Input elements in newly allocated memory
    printf("\n Enter %d new elements: \n", new_size - max);
    for(x = max; x < new_size; ++x){
        scanf(" %d", (ptr_numbers + x));
    }

    // Print all elements
    printf("\n Array elements are:");
    for (x = 0; x < new_size; ++x){
        printf("\n %d ", *(ptr_numbers + x));
    }

    // Release allocated memory
    free(ptr_numbers);

    return 0;
}
```

```
 Enter total number of elements: 5

 Enter 5 elements:
100
200
300
400
500

 Enter new size of the array: 8

 Enter 3 new elements:
600
700
800

 Array elements are:
 100
 200
 300
 400
 500
 600
 700
 800
```

Embedded Systems

C language

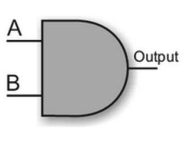


20- Bit Manipulation

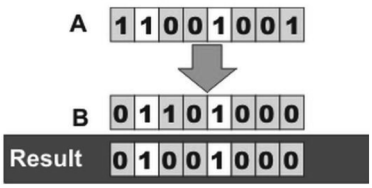
Eng. Mohamed Yousef

Logic Gates

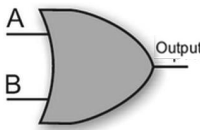
AND Gate



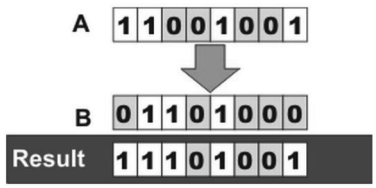
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



OR Gate



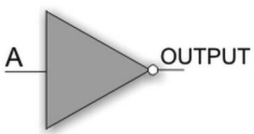
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



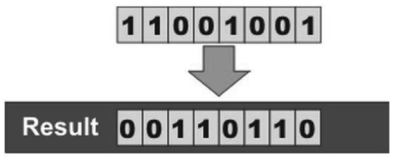
electronics010.blogspot.com C language Eng. Mohamed Yousef

Logic Gates

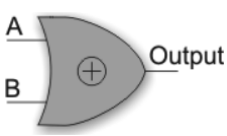
NOT Gate



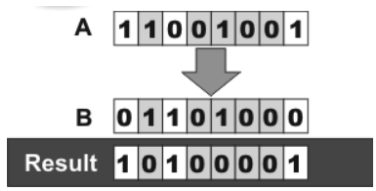
A	Output
0	1
1	0



XOR Gate



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0



Bitwise operators

- Data in the memory (RAM) is organized as a sequence of bytes.
- Each byte is a group of eight consecutive bits.
- We use *Bitwise* operators to manipulate data at its lowest level (bit level).
- *Bitwise* operators work with integer type. They do not support float types.

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

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bitwise AND & operator

Suppose `a` and `b` are two integer variables with initial value

```
int a=10, b=11;
```

Let us re-write integers in 8-bit binary representation

```
a = 0000 1010
b = 0000 1011
```

```
c = a & b
```

The above expression `a & b` will evaluate to `0000 1010` which is 10 in decimal.

```
a       0000 1010
b       0000 1011
-----------------
a & b   0000 1010
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bitwise OR | operator

Suppose `a` and `b` are two integer variables with initial value as.

```
int a=2, b=13;
```

Let us re-write the integer in 8-bit binary representation.

```
a = 0000 0010
b = 0000 1101
```

```
c = a | b
```

The above expression `a|b` will evaluate to `0000 1111` which is 15 in decimal.

```
a       0000 0010
b       0000 1101
-----------------
a | b   0000 1111
```

Bitwise complement ~ operator

Suppose `a` is an integer variable with initial value as.

```
int a=2;
```

In 8-bit binary representation:

```
a = 0000 0010
```

```
c = ~a
```

The above expression `~a` will evaluate to `1111 1101` which is -3 (Twos complement) in decimal.

```
a       0000 0010
-----------------
~a      1111 1101
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bitwise XOR ^ operator

Suppose `a` and `b` are two integer variables with initial value as.

```
int a=6, b=13;
```

Let us re-write the integers in 8-bit binary representation:

```
a = 0000 0110
b = 0000 1101
```

```
c = a ^ b
```

The above expression `a^b` will evaluate to `0000 1011` which is 11 in decimal.

```
a       0000 0110
b       0000 1101
-----------------
a ^ b   0000 1011
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Bitwise left shift << operator

Bitwise left shift is a binary operator. It is used to shift bits to left `n` times. Consider the below example:

```
int a=15;
```

Which in 8-bit binary will be represented as:

```
a = 0000 1111
```

```
c = a << 3;
```

The above expression `a << 3;` shifts bits of `a` three times to left and evaluates to `0111 1000` which is 120 in decimal.

```
a       0000 1111
a << 1  0001 1110
a << 2  0011 1100
a << 3  0111 1000
```

Bitwise left shift << operator

Important note: Shifting bits to left is also equivalent to multiplying value by 2. You can use bitwise left shift operator if you need to multiply a variable by a power of two.

Bitwise right shift >> operator

Bitwise right shift is binary operator used to shift bits to right. Consider the below example:

```
int a=15;
```

Which in 8-bit binary will be represented as:

```
a = 0000 1111
```

```
c = a >> 3
```

The above expression `a >> 3` shifts bits of variable `a` three times right and will evaluate to `0000 0001` which is 1 in decimal

```
a       0000 1111
a >> 1  0000 0111
a >> 2  0000 0011
a >> 3  0000 0001
```


Bitwise right shift >> operator

> ***Important note:*** *Shifting bits to right is equivalent to dividing by 2. You can use bitwise right shift operator if you need to divide a number (unsigned number) by power of 2.*

Bitwise right shift >> operator

Arithmetic Shift Right (Sign Extend)

```
1  x = -6;        // x = 0b11111010 = -6
2  y = x >> 2;    // y = 0b11111110 = -2
```

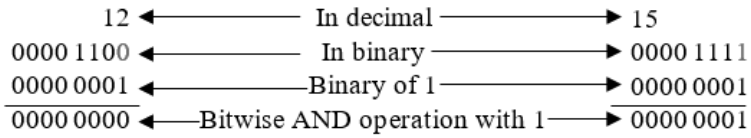
If the operand is SIGNED, a shift right performs an ARITHMETIC shift right, in which the vacant bits are filled with whatever the value of the most significant bit was before the shift. This is done to perform a sign extend when working with signed binary values

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

Write a C program to input any number from user and check whether the Least Significant Bit (LSB) of the given number is set (1) or not (0).

We use Bitwise AND & operator to check status of any bit.



As you can see in above image 12 & 1 evaluate to 0. Since, LSB of 12 is 0.
Whereas, 15 & 1 evaluate to 1 since LSB of 15 is 1.

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

```c
/*
 * C program to check Least Significant
   Bit (LSB) of a number using bitwise operator
 */

#include <stdio.h>

int main(){
int num;

    /* Input number from user */
    printf("\n Enter any number: ");
    scanf("%d", &num);

    /* If (num & 1) evaluates to 1 */
    if(num & 1)
        printf("\n LSB of %d is set (1).", num);
    else
        printf("\n LSB of %d is unset (0).", num);

    return 0;
}
```

```
Enter any number: 38

LSB of 38 is unset (0).
```

```
Enter any number: 9

LSB of 9 is set (1).
```

Example

Write a C program to input any number from user and check whether Most Significant Bit (MSB) of given number is set (1) or not (0).

1. Input a number from user. Store it in some variable say `num` .
2. Find number of bits required to represent an integer in memory. Use `sizeof()` operator to find size of integer in bytes. Then multiply it by 8 to get number of bits required by integer. Store total bits in some variable say `bits = sizeof(int) * 8;` .
3. To get MSB of the number, move first bit of 1 to highest order. Left shift 1 `bits - 1` times and store result in some variable say `msb = 1 << (bits - 1)` .
4. If bitwise AND operation `num & msb` evaluate to true then MSB of `num` is set otherwise not.

```
// Let int type = 1 byte
int num, bits, msb;
num = 0b10001101;
bits = sizeof(int) * 8
// bits = 1 * 8
bits = bits - 1 // 7
msb = 1 << bits
// msb = 0b00000001 << 7
// msb = 0b10000000

// num = 0b10001101 &
// msb = 0b10000000
//     = 0b10000000
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

```
/*
 * C program to check Most Significant Bit (MSB) of
   a number using bitwise operator
 */

#include <stdio.h>

int main(){
int num, bits, msb;

    // Input number from user
    printf("\n Enter any number: ");
    scanf("%d", &num);

    // Total bits required to represent integer
    bits = sizeof(int) * 8;

    // Move first bit of 1 to highest order
    msb = 1 << (bits - 1);

    // Perform bitwise AND with msb and num
    if(num & msb)
        printf("\n MSB of %d is set (1).", num);
    else
        printf("\n MSB of %d is unset (0).", num);

    return 0;
}
```

```
Enter any number: 5

MSB of 5 is unset (0).
```

```
Enter any number: -5

MSB of -5 is set (1).
```

Important note: Most Significant Bit of positive number is always `0` (in 2s complement) and negative number is `1` .

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

Write a C program to input any number from user and *toggle* or *invert* or *flip* nth bit of the given number using bitwise operator.

To toggle a bit we will use bitwise XOR ^ operator.

XOR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Step by step descriptive logic to toggle nth bit of a number.

1. Input number and nth bit position to toggle from user. Store it in some variable say `num` and `n` .
2. Left shift 1 to `n` times, i.e. `1 << n` .
3. Perform bitwise XOR with `num` and result evaluated above i.e. `num ^ (1 << n);` .

Example

```c
/*
 * C program to toggle nth bit of a number
 */

#include <stdio.h>

int main(){
int num, n, newNum;

    // Input number from user
    printf("\n Enter any number: ");
    scanf("%d", &num);

    /* Input bit position you want to toggle */
    printf("\n Enter nth bit to toggle (0-31): ");
    scanf("%d", &n);

    // Left shifts 1, n times
    // then perform bitwise XOR with num
    newNum = num ^ (1 << n);

    printf("\n Number before toggling %d.", num);
    printf("\n Number after  toggling %d.", newNum);

    return 0;
}
```

```
Enter any number: 45

Enter nth bit to toggle (0-31): 28

Number before toggling 45.
Number after  toggling 268435501.
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

1. How to set a bit in the number 'num' :

If we want to set a bit at nth position in number 'num' ,it can be done using 'OR' operator(|).

- First we left shift '1' to n position via (1<<n)
- Then, use 'OR' operator to set bit at that position.

OR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

```c
// C program to set nth bit of a number

#include <stdio.h>

int main(){
int num, n;

    // Input number from user
    printf("\n Enter any number: ");
    scanf("%d", &num);

    /* Input bit position you want to toggle */
    printf("\n Enter nth bit to set (0-31): ");
    scanf("%d", &n);

    // Left shifts 1, n times
    // then perform bitwise OR with num
    // newNum = num | (1 << n);
    // num = num | (1 << n);
    num |= (1 << n);

    printf("\n Number after  set %d.", num);

    return 0;
}
```

```
Enter any number: 4

Enter nth bit to set (0-31): 0

Number after  set 5.
```

```
Enter any number: 4

Enter nth bit to set (0-31): 1

Number after  set 6.
```

Example

2. How to unset/clear a bit at n'th position in the number 'num' :

Suppose we want to unset a bit at nth position in number 'num' then we have to do this with the help of 'AND' (&) operator.

- First we left shift '1' to n position via (1<<n) than we use bitwise NOT operator '~' to unset this shifted '1'.
- Now after clearing this left shifted '1' i.e making it to '0' we will 'AND'(&) with the number 'num' that will unset bit at nth position position.

```
// Let int type = 1 byte
int num, n;
num = 0b00000111;
n = 1;
// 1 << n
// => 0b00000001 << 1
// => 0b00000010
// => ~(0b00000010)
// =>   0b11111101

num = 0b00000111 &
      0b11111101
num = 0b00000101
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

Example

```
// C program to clear nth bit of a number

#include <stdio.h>

int main(){
int num, n;

    // Input number from user
    printf("\n Enter any number: ");
    scanf("%d", &num);

    /* Input bit position you want to toggle */
    printf("\n Enter nth bit to clear (0-31): ");
    scanf("%d", &n);

    // (1 << n)
    // ~(1 << n)
    // num = num & (~(1 << n));
    num &= ~(1 << n);

    printf("\n Number after clear %d.", num);

    return 0;
}
```

```
Enter any number: 7
Enter nth bit to clear (0-31): 1
Number after clear 5.
```

```
Enter any number: 8
Enter nth bit to clear (0-31): 3
Number after clear 0.
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

```
// C program to get value of  nth bit

#include <stdio.h>

int main(){
int num, n;

    // Input number from user
    printf("\n Enter any number: ");
    scanf("%d", &num);

    /* Input bit position you want to get */
    printf("\n Enter nth bit to (0-31): ");
    scanf("%d", &n);

    // num >> n
    // num & 1
    num = (num >> n) & 1;

    printf("\n Nth bit value %d.", num);

    return 0;
}
```

```
Enter any number: 8
Enter nth bit to (0-31): 3
Nth bit value 1.
```

```
Enter any number: 8
Enter nth bit to (0-31): 2
Nth bit value 0.
```

Bit Manipulation in C

Writing programs for embedded systems often requires the manipulation of individual bits. Below is a list of the most common operations. In the examples we are performing them on an unsigned long variable called **ctrl_reg**.

Setting a bit

We can set a bit using the bitwise OR operator (|):

```
ctrl_reg |= 1 << n;
```

The code above will set to 1 the bit in position **n**.

Clearing a bit

We can clear a bit using the bitwise AND operator (&) :

```
ctrl_reg &= ~(1 << n);
```

The code above will clear bit **n**. We first invert the bits with the bitwise NOT operator (~) and then perform the AND operation.

Toggling a bit

We can use the XOR operator (^) to toggle a bit.

```
ctrl_reg ^= 1 << n;
```

The code above will toggle bit **n**.

Getting the value of a bit

We can get the value of a particular bit by shifting and then using bitwise AND operation:

```
bit_value = (ctrl_reg >> n) & 1;
```

That will put the value of bit **n** into the variable **bit_value**.

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB

electronics010.blogspot.com C language Eng. Mohamed Yousef

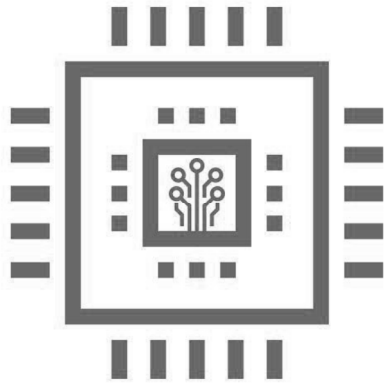
```
SET_BIT(num1, bit)
CLEAR_BIT(num2, bit)
TOGGLE_BIT(num3, bit)
GET_BIT(num4, bit)
```

```c
#include <stdio.h>

#define SET_BIT(reg, bit) (reg |= (1 << bit))
#define CLEAR_BIT(reg, bit) (reg &= ~(1 << bit))
#define TOGGLE_BIT(reg, bit) (reg^(1 << bit))
#define GET_BIT(reg, bit) ((reg >> bit) & 1)

int main(){
int num1 = 4, num2 = 7, num3 = 45, num4 = 8, bit = 3;

    printf("\n Number after setting: %d", SET_BIT(num1, bit));
    printf("\n Number after setting: %d", CLEAR_BIT(num2, bit));
    printf("\n Number after toggling: %d", TOGGLE_BIT(num3, bit));
    printf("\n Number after setting: %d", GET_BIT(num4, bit));

    return 0;
}
```

```
Number after setting: 12
Number after setting: 7
Number after toggling: 37
Number after setting: 1
```

Embedded Systems

C language



22- Common Arithmetic functions

Eng. Mohamed Yousef

abs() function

- abs( ) function in C returns the absolute value of an integer. The absolute value of a number is always positive. Only integer values are supported in C.
- "stdlib.h" header file supports abs( ) function in C language. Syntax for abs( ) function in C is given below.

```
int abs ( int n );
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int m = abs(200);
   int n = abs(-400);

   printf("Absolute value of m = %d\n", m);
   printf("Absolute value of n = %d \n",n);
   return 0;
}
```

OUTPUT:

```
Absolute value of m = 200
Absolute value of n = 400
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

floor() function

- floor( ) function in C returns the nearest integer value which is less than or equal to the floating point argument passed to this function.
- "math.h" header file supports floor( ) function in C language. Syntax for floor( ) function in C is given below.

```
double floor ( double x );
```

```c
 #include <stdio.h>
#include <math.h>
 int main()
{
       float i=5.1, j=5.9, k=-5.4, l=-6.9;
       printf("floor of  %f is  %f\n", i, floor(i));
       printf("floor of  %f is  %f\n", j, floor(j));
       printf("floor of  %f is  %f\n", k, floor(k));
       printf("floor of  %f is  %f\n", l, floor(l));
       return 0;
}
```

OUTPUT:

```
floor of 5.100000 is 5.000000
floor of 5.900000 is 5.000000
floor of -5.400000 is -6.000000
floor of -6.900000 is -7.000000
```

ceil() function

- ceil( ) function in C returns nearest integer value which is greater than or equal to the argument passed to this function.
- "math.h" header file supports ceil( ) function in C language. Syntax for ceil( ) function in C is given below.

```
double ceil (double x);
```

```c
#include <stdio.h>
#include <math.h>
int main()
{
        float i=5.4, j=5.6;
        printf("ceil of  %f is  %f\n", i, ceil(i));
        printf("ceil of  %f is  %f\n", j, ceil(j));
        return 0;
}
```

OUTPUT:

```
ceil of 5.400000 is 6.000000
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

round() function

- round( ) function in C returns the nearest integer value of the float/double/long double argument passed to this function.
- If decimal value is from ".1 to .4 it returns integer value less than the argument. If decimal value is from .5 :o .9", it returns the integer value greater than the argument.
- "math.h" header file supports round( ) function in C language. Syntax for round( ) function in C is given below.

```
double round (double a);
```

```c
#include <stdio.h>
#include <math.h>
int main()
{
        float i=5.4, j=5.6;
        printf("round of  %f is  %f\n", i, round(i));
        printf("round of  %f is  %f\n", j, round(j));
        return 0;
}
```

OUTPUT:

```
round of 5.400000 is 5.000000
round of 5.600000 is 6.000000
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

C library function - rand()

The C library function **int rand(void)** returns a pseudo-random number in the range of 0 to *RAND_MAX*.

RAND_MAX is a constant whose default value may vary between implementations but it is granted to be at least 32767.

```
int rand(void)
```

```c
#include <stdio.h>
#include <stdlib.h>
int main(){

   printf("\n Random value: %d", rand());
   printf("\n Random value: %d", rand());

    return 0;
}
```

```
Random value: 41
Random value: 18467
```

C library function - rand()

```
r = rand() % (upper limit - lower limit + 1) + lower limit
```

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

int main(){
int x  = 0, r = 0;
const int MAX = 7;
const int MIN = 3;

    for(x = 0; x < 10; ++x){
        r = rand() % (MAX - MIN + 1) + MIN;
        printf("\n Random value: %d", r);
    }

    return 0;
}
```

```
Random value: 4
Random value: 5
Random value: 7
Random value: 3
Random value: 7
Random value: 7
Random value: 6
Random value: 6
Random value: 5
Random value: 7
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

srand() function

The srand() function sets the starting point for producing a series of pseudo-random integers. If srand() is not called, the rand() seed is set as if srand(1) were called at program start. Any other value for seed sets the generator to a different starting point.

```
void srand(unsigned int seed)
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

srand() function

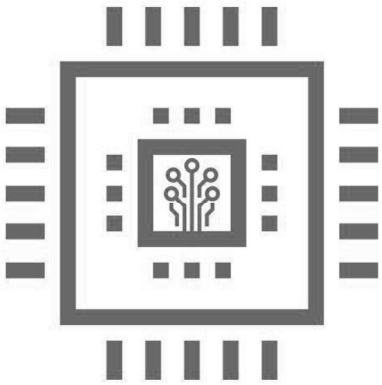
```c
#include <stdio.h>
#include <stdlib.h>

int main(){
int x  = 0, r = 0, see = 0;
const int MAX = 7;
const int MIN = 3;

    printf("\n Enter seed:");
    scanf("%d", &see);

    srand(see);

    for(x = 0; x < 10; ++x){
        r = rand() % (MAX - MIN + 1) + MIN;
        printf("\n Random value: %d", r);
    }

    return 0;
}
```

```
Enter seed:123

Random value: 3
Random value: 6
Random value: 3
Random value: 7
Random value: 6
Random value: 3
Random value: 7
Random value: 6
Random value: 3
Random value: 7
```

```
Enter seed:5

Random value: 7
Random value: 6
Random value: 3
Random value: 7
Random value: 3
Random value: 3
Random value: 5
Random value: 7
Random value: 5
Random value: 3
```

Embedded Systems

C language



22- Manipulating String

Eng. Mohamed Yousef

String

- C Strings are nothing but array of characters ended with null character (‘\0’).
- This null character indicates the end of the string.
- Strings are always enclosed by double quotes. Whereas, character is enclosed by single quotes in C.

C STRING FUNCTIONS:

- String.h header file supports all the string functions in C language.

All the string functions are given below.

https://fresh2refresh.com/c-programming/c-strings/

electronics010.blogspot.com C language Eng. Mohamed Yousef

strlen() function

- strlen( ) function in C gives the length of the given string. Syntax for strlen( ) function is given below.

```
strlen ( const char * str );
```

- strlen( ) function counts the number of characters in a given string and returns the integer value.
- It stops counting the character when null character is found. Because, null character indicates the end of the string in C.

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

int main(){
char a[20] = "Program";
char b[20] = {'P','r','o','g','r','a','m','\0'};
char c[20] = {'\0'};

    printf("\n Length of string a = %d",strlen(a));
    printf("\n Length of string b = %d",strlen(b));
    printf("\n Length of string c = %d",strlen(c));

    return 0;
}
```

```
Length of string a = 7
Length of string b = 7
Length of string c = 0
```

strcpy() function

- strcpy( ) function copies contents of one string into another string. Syntax for strcpy function is given below.

```
char *strcpy(char *dest, const char *src)
```

- **dest** – This is the pointer to the destination array where the content is to be copied.
- **src** – This is the string to be copied.

This returns a pointer to the destination string dest.

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

int main(){

char source[] = "Embedded Systems" ;
char target[20] = {'\0'};

    printf ( "\n Target string: %s", strcpy ( target, source )) ;

    return 0;
}
```

```
Target string: Embedded Systems
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

strcat() function

- strcat( ) function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for strcat( ) function is given below.

```
char *strcat(char *dest, const char *src)
```

- **dest** – This is pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string.
- **src** – This is the string to be appended.

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

int main(){

char target[] = "Embedded ";

    printf ("\n Target string: %s", strcat(target, "Systems"));

    return 0;
}
```

```
Target string: Embedded Systems
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

strcmp() function

- strcmp( ) function in C compares two given strings

Syntax for strcmp( ) function is given below.

```
int strcmp(const char *str1, const char *str2)
```

Return Value from strcmp()

Return Value	Remarks
0	if both strings are identical (equal)
negative	if the ASCII value of first unmatched character is less than second.
positive integer	if the ASCII value of first unmatched character is greater than second.

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

int main(){
char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
int result;

    result = strcmp(str1, str2);
    printf("\n strcmp(str1, str2) = %d", result);

    result = strcmp(str2, str1);
    printf("\n strcmp(str2, str1) = %d", result);

    result = strcmp(str1, str3);
    printf("\n strcmp(str1, str3) = %d", result);

    return 0;
}
```

```
strcmp(str1, str2) = 1
strcmp(str2, str1) = -1
strcmp(str1, str3) = 0
```

electronics010.blogspot.com C language Eng. Mohamed Yousef

https://fresh2refresh.com/c-programming/c-strings/

https://fresh2refresh.com/c-programming/c-int-char-validation/

electronics010.blogspot.com C language Eng. Mohamed Yousef

LAB