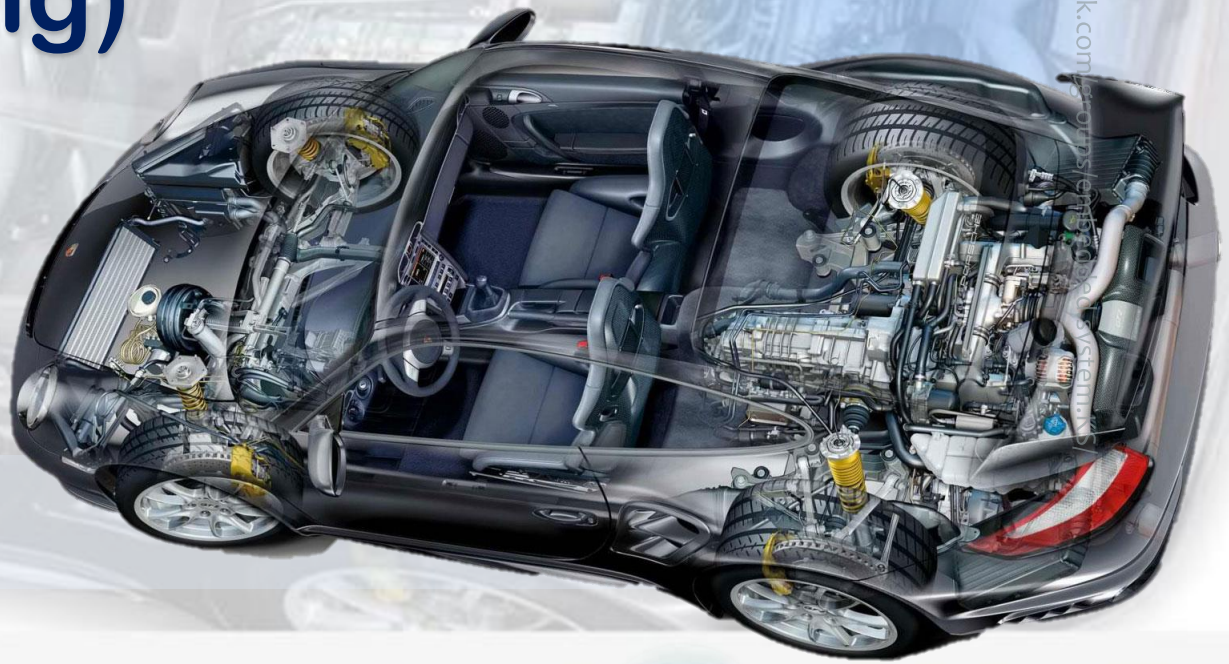
#LEARN_IN DEPTH

#Be_professional_in embedded_system

Embedded System
PART 4(C Programming)

- difference between variable definition and declaration
- C Functions
- C storage classes
- Inline assembly
- Inline function



ENG.KEROLES SHENOUDA

difference between variable definition and declaration

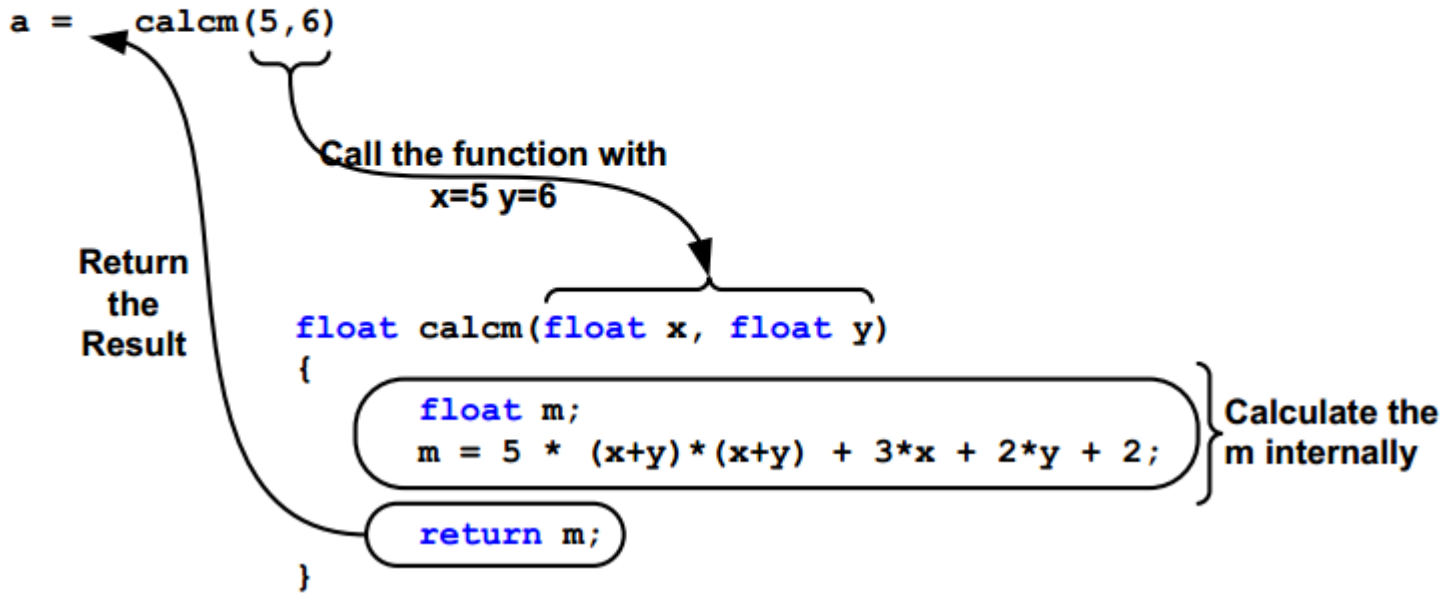
- **definition**
  - a variable is defined when the **compiler** generates instructions **to allocate the storage** of the variable
- **declaration**
  - a variable is declared when the **compiler** is informed that **a variable exists along with its type**. The compiler does not generate instructions to allocate the storage for the variable a that point.
  - Variable can be **declared** many times in a program. But, **definition** can happen **only one time** for a variable in a program.
- **A variable definition is also a declaration, but not all variable declarations are definitions**
  - **Extern int x ; //declaration without definition,** The variable was already defined somewhere else and it's just recalled here
  - **int x ; //definition and declaration**

Functions

```c
#include "stdio.h"
#include "math.h"

float calcm(float x, float y)//Function
{
	float m;
	m = 5 * (x+y)*(x+y) + 3*x + 2*y + 2;
	return m;
}

void main()
{
	float z;

	z = (calcm(3,2) +
	sqrt(pow(calcm(6,1.5), 3.2f) +
				calcm(5,3.4)))/calcm(13,1.2);

	printf("z = %f\r\n", z);
}
```

Understanding `calcm` function:

`calcm` function is a small program, simply it takes the values of x and y, calculating m internally, then return the calculated value.



The call **`a = calcm(5,6)`** works as follows:

1. Copies the values 5 and 6 to the variables x and y
2. Performs the internal calculations to calculate m
3. When executing the line (**return m**), the computer copies the value inside m and return it to the line (**a = calcm(5,6)**)
4. Copies the return value to (**a**) variable

Function Definition

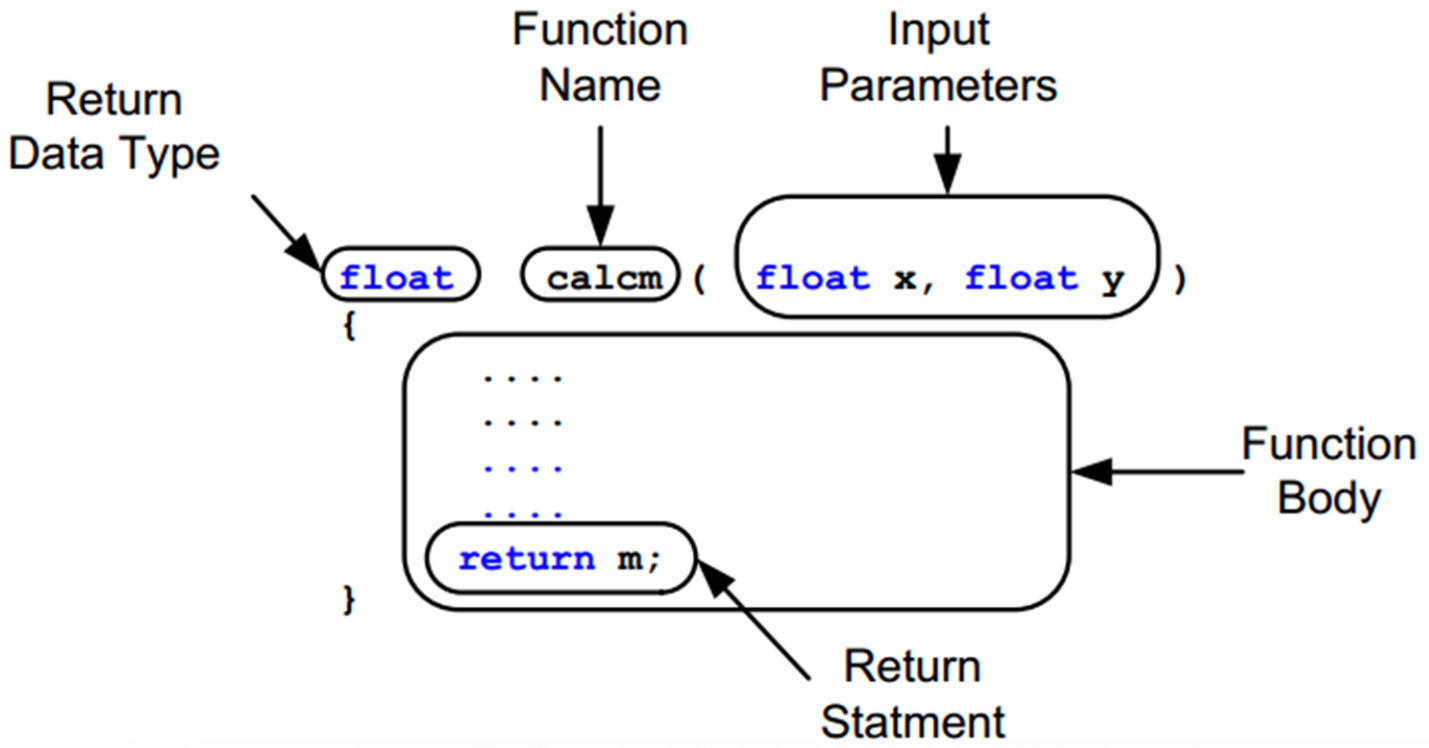
Function Name: like variable, must have no spaces and no special characters and must starts with letters

Input Parameters: supplied parameters types and names. You can define any number of inputs; also you can define zero number of inputs.

Return Type: the data type of the function output, if the function has no output use (**void**) keyword.

Function Body: performs specific function operation.

Return Statement: this statement tells the computer that the function execution is completed and the required function output is ready for the caller. The computer takes the returned value and supplies it to the caller.



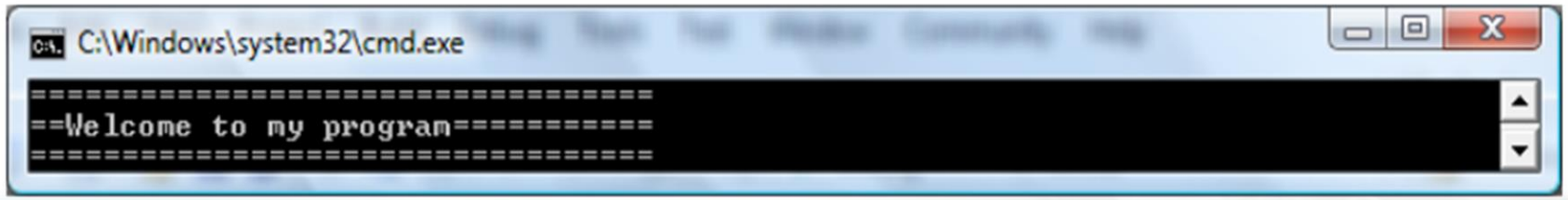
Prototype

```c
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

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

    functionName();

    ... .. ...
    ... .. ...
}
```

```c
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

Important: sometimes functions does not take any input and does not return an output, this means that the function performs only an internal operation. Following example clarifies this idea.

```
#include "stdio.h"

void printWelcome()
{
	printf("=====================================\r\n");
	printf("=======Welcome to my program=======\r\n");
	printf("=====================================\r\n");
}

void main()
{
	printWelcome();
}
```



C functions aspects	syntax
function definition	Return_type function_name (arguments list) { Body of function; }
function call	function_name (arguments list);
function declaration	return_type function_name (argument list);

Where to define the function? To answer these questions try the following program.

```
#include "stdio.h"

void main()
{
	printWelcome();
}

void printWelcome()
{
	printf("=======================================\r\n");
	printf("=======Welcome to my program=======\r\n");
	printf("=======================================\r\n");
}
```

The compiler gives an **error** at the line **`printWelcome();`** in the main function, the error state that "the function **`printWelcome`** is **undefined**". Which means that the compiler cannot locate the function before the main, even if it is located after the main?

To solve this problem, you have to move the function before the main, or to move only the function prototype, as shown in the following program.

```
#include "stdio.h"

//Function Prototype
void printWelcome();

void main()
{
	printWelcome();
}

void printWelcome()
{
	printf("=======================================\r\n");
	printf("=======Welcome to my program=======\r\n");
	printf("=======================================\r\n");
}
```

Calculate the Factorial

- Write a program uses a function to calculate the factorial of any positive number

Factorial of x means $x_! = (x) \times (x-1) \times (x-2) \ldots \times (3) \times (2) \times (1)$
For example: $5_! = 5 \times 4 \times 3 \times 2 \times 1$
Specially $0_! = 1$

Problems Tasks Console Properties AVR Device Explo
<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Cou

```
Factorial(10) = 3628800

Factorial(0) = 1

Factorial(5) = 120
```

Lab

main.c

```c
/*
 * main.c
 *
 *  Created on: Mar 23, 2017
 *      Author: Keroles
 */
#include <stdio.h>
#include <string.h>


int factorial(int x)
{
    int f = 1;
    for(;x>0;x--)
        f *= x;
    return f;
}
int main()
{

    printf("Factorial(%d) = %d\r\n", 10, factorial(10));
    printf("Factorial(%d) = %d\r\n", 0, factorial(0));
    printf("Factorial(%d) = %d\r\n", 5, factorial(5));

    return 0 ;
}
```


#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Lab

Solution

Lab:Calculate the Minimum Value of any Given Array

```
int calcMin(int values[], int n) ;
```

Important: **calcMin** function takes two parameters an array and (**int**) value containing the array size. Know that function could not expect the given array size, you must supply it by yourself.

Lab


```c
#include "stdio.h"

int calcMin(int values[], int n)
{
	int i, minvalue = values[0];
	for(i=0; i<n; i++)
	{
		if(values[i]<minvalue)
			minvalue = values[i];
	}
	return minvalue;
}

void main()
{
	int xvalues[10] = {35, 67, 27, 54, 76,
					44, 59, 32, 43, 25};
	int yvalues[5] = {28, 71, 67, 83, 62};
	int zvalues[13] = {87, 21, 74, 36, 27,
				64, 87, 63, 27, 86, 48, 32, 76};

	printf("The minimum of x, y, z values is
			(%d, %d, %d)\r\n", calcMin(xvalues,10 ),
			calcMin(yvalues, 5), calcMin(zvalues, 13));
}
```

Lab

Solution

Finding a Name in a Set of Names

```
int findName(char names[][14], int n, char name[]);
```

```
void main()
{
char name[14];
char names[5][14] = {"Alaa", "Osama", "Mamdouh",
"Samy", "Hossain"};
puts("Enter your first name:");
gets(name);
if(findName(names, 5, name)==1)
puts("Welcome");
else
puts("Goodby");
}
```

Lab

Difference between Passing Single Values and Arrays

Pass by value

This method copies **the actual value** of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Pass by reference

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

main.c

```c
/*
 * main.c
 *
 *  Created on: Mar 23, 2017
 *      Author: Keroles
 */
#include <stdio.h>
#include <string.h>

void tryToModify(int x, char text[])
{
    x++;
    text[0]--;
}
int main()
{
    int v = 5;
    char name[5] = "Good";
    printf("v = %d, name = %s\r\n", v, name);
    tryToModify(v, name);
    printf("v = %d, name = %s\r\n", v, name);

    return 0 ;
}
```

Problems | Tasks | Console | Properties | AVR Device Explorer | AVR Supported MCUs

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Course\session2\Debug\session2.exe (3/31/17, 11:00 AM

```
v = 5, name = Good

v = 5, name = Food
```

FOLLOW US

Press here

17

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Pass by value

x

Pass by reference

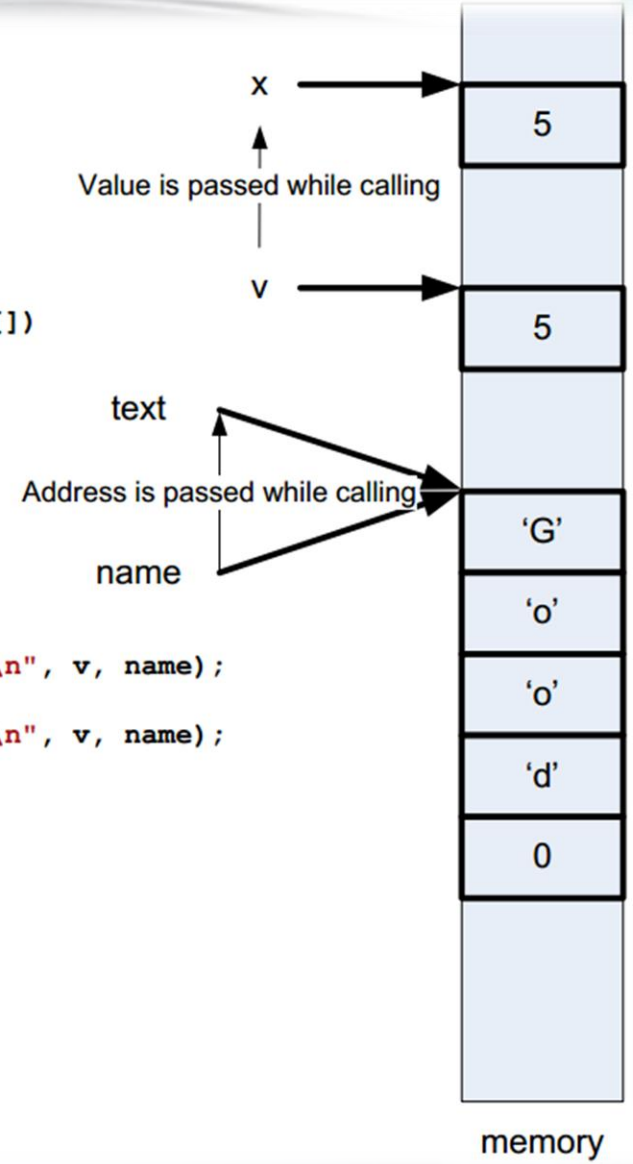
Text[]

https://www.facebook.com/groups/embedded.system.KS/

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

```c
#include "stdio.h"

void tryToModify(int x, char text[])
{
	x++;
	text[0]--;
}

void main()
{
	int v = 5;
	char name[5] = "Good";
	printf("v = %d, name = %s\r\n", v, name);
	tryToModify(v, name);
	printf("v = %d, name = %s\r\n", v, name);
}
```



#LEARN_IN DEPTH

#Be_professional_in_embedded_system

(x)= Variables | Breakpoints | Registers | Modules

Name	Type	Value
(x)= v	int	5
name	char [5]	0x61ff27
(x)= name[0]	char	71 'G'
(x)= name[1]	char	111 'o'
(x)= name[2]	char	111 'o'
(x)= name[3]	char	100 'd'
(x)= name[4]	char	0 '\0'

main.c

```
12      x++;
13      text[0]--;
14  }
15  int main()
16  {
17      int v = 5;
18      char name[5] = "Good";
19      printf("v = %d, name = %s\r\n", v, name);
20      tryToModify(v, name);
21      printf("v = %d, name = %s\r\n", v, name);
```

File Edit Source Refactor Navigate Search Project AVR Run

Debug

- session2.exe [C/C++ Application]
  - session2.exe [4548]
    - Thread #1 0 (Suspended : Step)
      - tryToModify() at main.c:12 0x401463
      - main() at main.c:20 0x4014cd
    - Thread #2 0 (Suspended : Container)
  - gdb (7.6.1)

main.c

```
10 void tryToModify(int x, char text[])
11 {
12     x++;
13     text[0]--;
14 }
15 int main()
16 {
17     int v = 5;
18     char name[5] = "Good";
19     printf("v = %d, name = %s\r\n", v, name);
20     tryToModify(v, name);
21     printf("v = %d, name = %s\r\n", v, name);
22
23     return 0 ;
24 }
25
```

(x)= Variables | Breakpoints | 1010 Registers | Modules

Main Stack

Name	Type	Value
(x)= v	int	5
name	char [5]	0x61ff27
(x)= name[0]	char	71 'G'
(x)= name[1]	char	111 'o'
(x)= name[2]	char	111 'o'
(x)= name[3]	char	100 'd'
(x)= name[4]	char	0 '\0'

Address name array on main() stack

Debug

- session2.exe [C/C++ Application]
  - session2.exe [4548]
    - Thread #1 0 (Suspended : Step)
      - tryToModify() at main.c:12 0x401463
      - main() at main.c:20 0x4014cd
    - Thread #2 0 (Suspended : Container)
  - gdb (7.6.1)

main.c

```
10 void tryToModify(int x, char text[])
11 {
12     x++;
13     text[0]--;
14 }
15 int main()
16 {
17     int v = 5;
18     char name[5] = "Good";
19     printf("v = %d, name = %s\r\n", v, name);
20     tryToModify(v, name);
21     printf("v = %d, name = %s\r\n", v, name);
22
23     return 0 ;
24 }
25
```

tryToModify Stack

Name	Type	Value
(x)= x	int	5
text	char *	0x61ff27 "Good"
(x)= *text	char	71 'G'

Address name array on main() stack

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

main.c

```
10 void tryToModify(int x, char text[])
11 {
12     x++;
13     text[0]--;
14 }
15 int main()
16 {
17     int v = 5;
18     char name[5] = "Good";
19     printf("v = %d, name = %s\r\n", v, name);
20     tryToModify(v, name);
21     printf("v = %d, name = %s\r\n", v, name);
22
23     return 0 ;
24 }
25
```

Debug

- session2.exe [C/C++ Application]
  - session2.exe [4548]
    - Thread #1 0 (Suspended : Step)
      - tryToModify() at main.c:14 0x401477
      - main() at main.c:20 0x4014cd
    - Thread #2 0 (Suspended : Container)
  - gdb (7.6.1)

The x changed on tryToModify() stack

(x)= Variables | Breakpoints | Registers | Modules

Name	Type	Value
(x)= x	int	6
text	char *	0x61ff27 "Food"
(x)= *text	char	70 'F'

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Debug

- session2.exe [C/C++ Application]
  - session2.exe [4548]
    - Thread #1 0 (Suspended : Step)
      - main() at main.c:21 0x4014cd
    - Thread #2 0 (Suspended : Container)
  - gdb (7.6.1)

The tryToModify() stack is removed

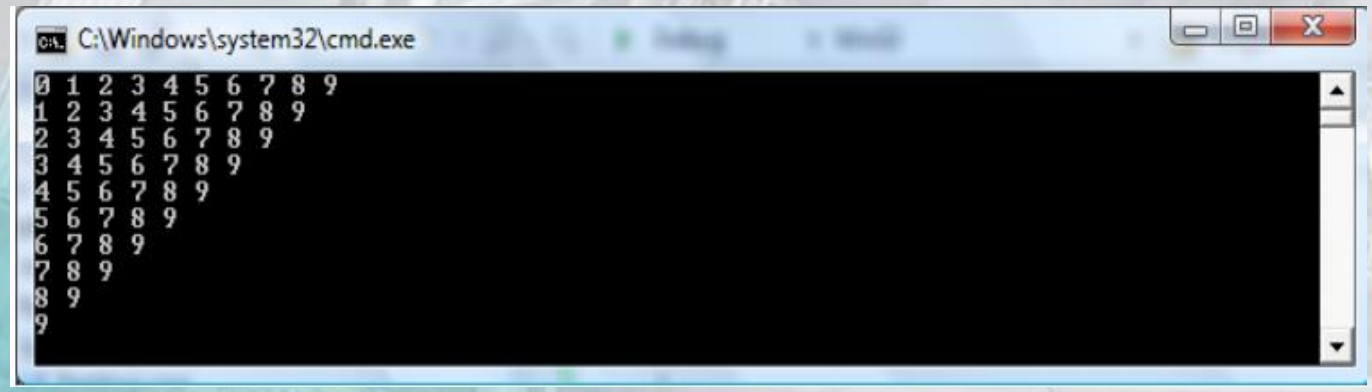
main.c

```
11 {
12     x++;
13     text[0]--;
14 }
15 int main()
16 {
17     int v = 5;
18     char name[5] = "Good";
19     printf("v = %d, name = %s\r\n", v, name);
20     tryToModify(v, name);
21     printf("v = %d, name = %s\r\n", v, name);
22
23     return 0 ;
24 }
25
```

(x)= Variables | Breakpoints | Registers | Modules

Name	Type	Value
(x)= v	int	5
name	char [5]	0x61ff27
(x)= name[0]	char	70 'F'
(x)= name[1]	char	111 'o'
(x)= name[2]	char	111 'o'
(x)= name[3]	char	100 'd'
(x)= name[4]	char	0 '\0'

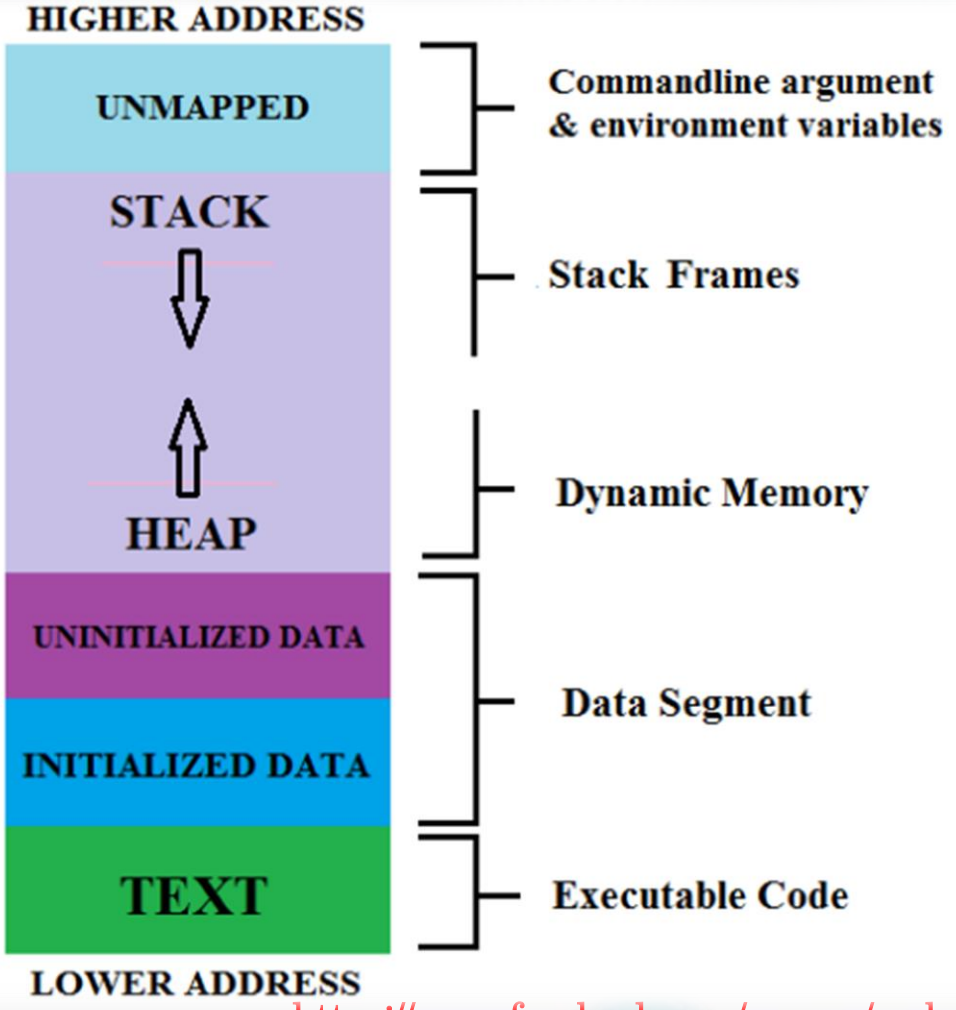
Write a program that produces the following output:



Lab

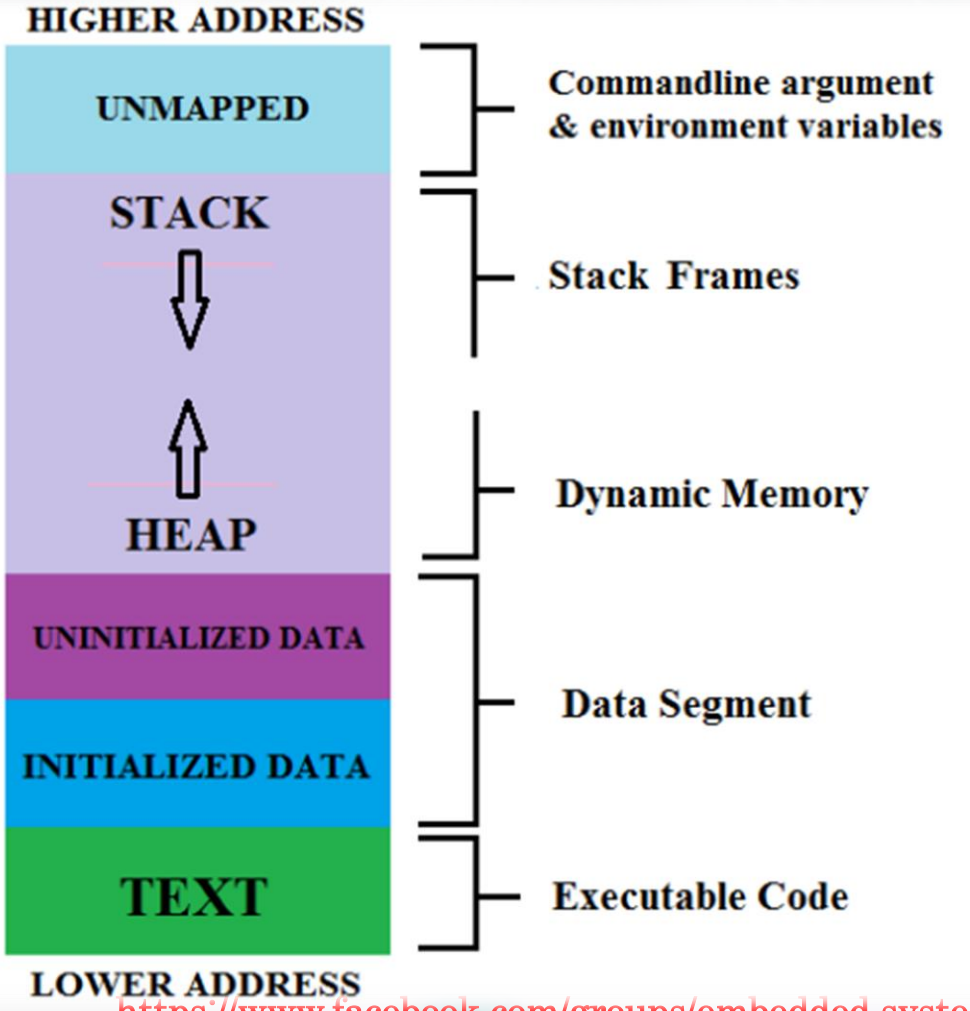
What is memory layout of C-program ?

- when you run any C-program, its executable image is loaded into RAM
  - This memory layout is organized in following fashion:
  - Text segment
  - **Data segment**
  - **Heap segment**
  - **Stack segment**
  - Unmapped or reserved



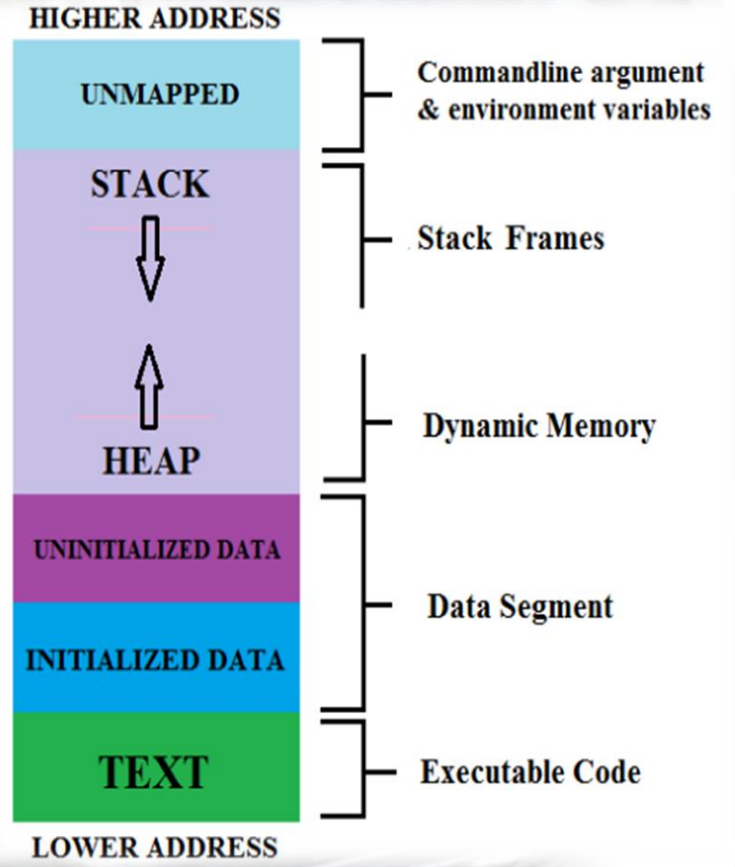
Text segment

- Text segment contain **executable instructions** of your C program, its also called **code segment**. This is the machine language representation of the program steps to be carried out, including all functions making up the program, both user defined and system



Data segment

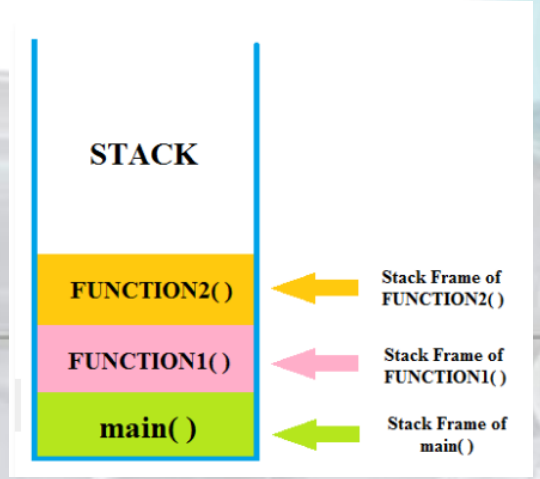
- There are two sub section of this segment called **initialized** & **uninitialized** data segment
  - **Initialized data:-** It contains both **static** and **global** data that are initialized with non-zero values.
    - This segment can be further classified into *read-only* area and *read-write* area.
    - For example : The global string defined by char string[ ] = "hello world" and a statement like int count=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global statement like const int A=3 makes the variable 'A' read-only and to be stored in initialized read-only area.
  - **Uninitialized data (bss segment):-** Uninitialized data segment is also called BSS segment. BSS stands for 'Block Started by Symbol' named after an ancient assembler operator. Uninitialized data segment contains all global and static variables that are initialized to zero or do not have explicit initialization in source code.
    - For example : The global variable declared as int A would be stored in uninitialized data segment. A statement like static int X=0 will also stored in this segment cause it initialized with zero.



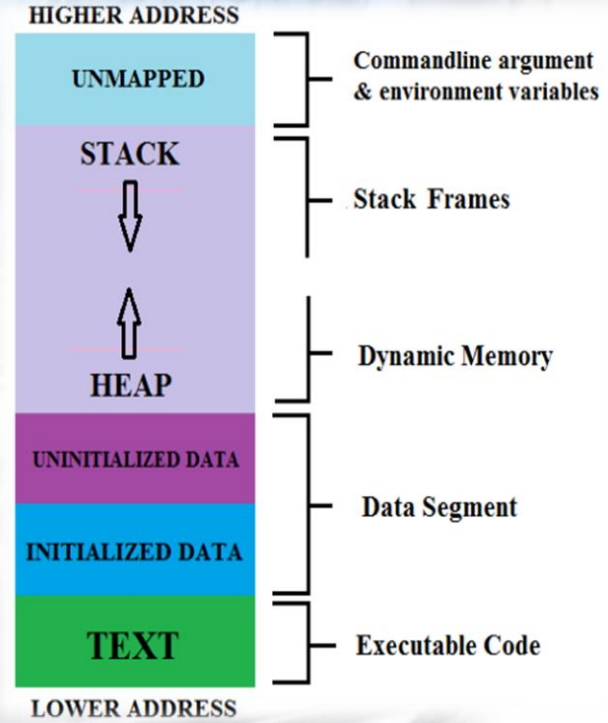
Heap segment

- The heap segment is area where dynamically allocated memory (allocated by malloc(), calloc(), realloc() and new for C++) resides.
- When we allocate memory through dynamic allocation techniques(in simple word, run time memory allocation), program acquire space from system and process address space grows that's why we saw upward arrow indication in figure for Heap.

Stack segment



- The stack segment is area where **local variables** are stored. By saying local variable means that all those variables which are declared in every function including main( ) in your C program.
- When we call any function, **stack frame is created** and when function returns, **stack frame is destroyed** including all local variables of that particular function.
- Stack frame **contain** some data like **return address, arguments passed** to it, **local variables**, and any **other information needed by the invoked function**.
- A **"stack pointer (SP)"** keeps track of stack by each push & pop operation onto it, by adjusted stack pointer to next or previous address.



Storage Classes in C

increasing **focus** and **concentration**

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Storage Class in C

```
storage_class var_data_type var_name;
```


- The storage classes in C determine the
  - Scope of a variable
  - Life time of a variable
  - Visibility of a variable or function
- C Storage classes:.
  - Automatic
  - External
  - Static
  - Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM (Stack)	Garbage Value	Local (within block)	Within function (end of block)
extern	RAM (Data segment)	Zero	Global (multiple files)	Till the end of the main program Maybe declared anywhere in the program
static	RAM (Data segment)	Zero	Local (within block)	Till the end of the main program, Retains value between multiple functions call
register	CPU general purpose Register	Garbage Value	Local (within block)	Within the function (within block)

Storage Class in C (auto)

- Automatic variables are allocated memory automatically at runtime.

- This is the **default storage** class for all the variables **declared inside a function or a block**.
- **The visibility** of the automatic variables **is limited to the block** in which they are defined.
- **The scope** of the automatic variables **is limited to the block** in which they are defined.
- Every **local variable** is **automatic** in C by default.
- Also it can be accessed **outside their scope** as well using the concept of **pointers** given here by pointing to the very exact memory location where the variables resides.
- The automatic variables **are initialized** to **garbage by default**.

Local Variables

Above program have three functions each with different scopes; each scope holds different local variables. The variable **a**, **b** and **c** of main function are inaccessible through **myAdd** or **myMull** functions. The variables **x**, **y** and **z** of **myMull** function are inaccessible through **myAdd** or **main** functions, even if **myAdd** function has the same variables names

```c
#include "stdio.h"

int myMull(int x, int y)
{
	int z;
	z = x * y;
	return z;
}

int myAdd(int x, int y)
{
	int z;
	z = x + y;
	return z;
}

void main()
{
	int a = 5, b = 6;
	printf("a + b = %d\r\n", myAdd(a,b));
	printf("a * b = %d\r\n", myMull(a,b));
}
```

myMull Scope
Local variables:
x,y,z

myAdd Scope
Local variables:
x,y,z

main Scope
Local variables:
a,b,c

Storage Class in C (auto)

```c
/*
 * main.c
 *
 *  Created on: Sep 5, 2020
 *      Author: kkhalil
 */

#include <stdio.h>
int main()
{
    int a = 10;
    printf("%d ",++a);
    {
        int a = 20;
        printf("%d ",a);
    }
    printf("%d ",a);
    return 0 ;
}
```

WHAT IS THE OUTPUT ? WHY?

Storage Class in C (auto)

```c
/*
 * main.c
 *
 *  Created on: Sep 5, 2020
 *      Author: kkhalil
 */

#include <stdio.h>
int main()
{
    int a = 10;
    printf("%d ",++a);
    {
        int a = 20;
        printf("%d ",a);
    }
    printf("%d ",a);
    return 0 ;
}
```

```
Problems  Tasks  Console  Properties
<terminated> (exit value: 0) test.exe [C/C++ Application] D:\courses\new_diploma\Diploma onlir
11 20 11
```

The visibility of the automatic variables **is limited to the block** in which they are defined.

The scope of the automatic variables **is limited to the block** in which they are defined.

Storage Class in C (static)

- The variables defined as static specifier **can hold their value** between the multiple function calls as it is stored on Data Memory.
- Static local variables **are visible only** to **the function or the block** in which they are defined.
- **A same static variable** can be **declared** **many times** but **can be assigned at only one time.**
- Default initial value of the static integral variable **is 0** otherwise null.
- **The visibility** of the **static global variable** is **limited to the file in which it has declared.**
- The keyword used to define static variable is **static**

Static Variables

- Static variables are defined by the modifier **static.** Static variables are initialized once in the program life. For example if the variable (**x**) is defined inside a function, the variable is initialized only at first function call, further function calls do not perform the initialization step, this means that if the variable is modified by any call the modification result remains for the next call. Following example illustrate this idea

```
#include "stdio.h"

void myprint()
{
	static int x = 0;
	int y = 0;
	x++;
	y++;
	printf("x = %d, y = %d\r\n", x, y);
}

void main()
{
	int i;
	for(i=0;i<10;i++)
		myprint();
```

Static Variables

Static for global Variables

stdio.h | file1.c | main.c

```c

int xxx=1 ;

void file1_func1(void)
{
    printf ("file1_func1 xxx=%d \n",xxx);
    xxx +=3 ;

}
```

stdio.h | file1.c | main.c

```c
/*
// * main.c
// *
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"
 static int xxx ;

void file1_func1(void);

int  main ()
{
    printf ("main.c xxx=%d \n",xxx);
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);
    xxx = 8 ;
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);

    return 0 ;
}
```

WHAT IS THE OUTPUT ? WHY?

FOLLOW US

Press here

39

#LEARN_IN DEPTH

#Be_professional_in embedded_system

https://www.facebook.com/groups/embedded.system.KS/

Static for global Variables

stdio.h | file1.c | main.c

```c

int xxx=1 ;

void file1_func1(void)
{
    printf ("file1_func1 xxx=%d \n",xxx);
    xxx +=3 ;

}
```


#LEARN_IN DEPTH

#Be_professional_in embedded_system

stdio.h | file1.c | main.c

```c
/*
// * main.c
// *
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"
 static int xxx ;

void file1_func1(void);

int  main ()
{
    printf ("main.c xxx=%d \n",xxx);
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);
    xxx = 8 ;
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);

    return 0 ;
}

```

<terminated> (exit value: 0) C_Programming.exe [C/C++ Application] D:

```
main.c xxx=0
file1_func1 xxx=1
main.c xxx=0
file1_func1 xxx=4
main.c xxx=8
```

The visibility of the **static global variable** is **limited to the file in which it has declared.**

https://www.facebook.com/groups/embedded.system.KS/

Storage Class in C (extern)

- The **external storage** class is used to **tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.**
- The variables declared as extern **are not allocated any memory**. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- ***The default initial*** value of external integral type **is 0** otherwise null.
- **We can only initialize the extern variable globally**
- **An external variable can be declared many times but can be initialized at only once.**
- If a variable is declared as **external** then the compiler searches for that variable to be initialized somewhere in the program, If it is not, then the compiler will show an error.

Global Variables

- In the following program the variable name is defined as a global variable, all program
function can access this variable.

```
#include "stdio.h"

char gName[50];

void welcome()
{
	printf("Welcome %s\r\n", gName);
}
void goodby()
{
	printf("Goodby %s\r\n", gName);
}

void main()
{
	puts("Enter your name:");
	gets(gName);
	welcome();
	goodby();
}
```

Global Variables

file1.c main.c

(Global Scope)

```c
#include "stdio.h"
 int xxx =7  ;
void file1_func1(void);

int  main ()
{
    printf ("main.c xxx=%d \n",xxx);
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);
    xxx = 8 ;
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);

    return 0 ;
}
```

file1.c main.c

(Global Scope)

```c
int xxx=1 ;

void file1_func1(void)
{
    printf ("file1_func1 xxx=%d \n",xxx);
    xxx +=3 ;

}
```

WHAT IS THE OUTPUT ? WHY?

FOLLOW US

Press here

43

#LEARN_IN DEPTH

#Be_professional_in embedded_system

file1.c main.c

(Global Scope)

```c
#include "stdio.h"
 int xxx =7  ;
void file1_func1(void);

int  main ()
{
    printf ("main.c xxx=%d \n",xxx);
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);
    xxx = 8 ;
    file1_func1 () ;
    printf ("main.c xxx=%d \n",xxx);

    return 0 ;
}
```

file1.c main.c

(Global Scope)

```c
int xxx=1 ;

void file1_func1(void)
{
    printf ("file1_func1 xxx=%d \n",xxx);
    xxx +=3 ;

}
```

So, if `x` is global

```
int x;
```

then its storage duration, scope and linkage is equivalent to `x` in

```
extern int x;
```

An external variable can be declared many times but can be initialized at only once.

Output

Show output from: Build

```
1>------ Build started: Project: Diploma_online, Configuration: Debug Win32 ------
1>Compiling...
1>main.c
1>Linking...
1>main.obj : error LNK2005: _xxx already defined in file1.obj
1>C:\Users\Public\Documents\Vector CANape 15\Extras\C-RT Project\Debug\Diploma_online.exe : fatal error LNK1169: one or more multiply defined symbols found
1>Build log was saved at "file://c:\Users\Public\Documents\Vector CANape 15\Extras\Diploma_online\Debug\BuildLog.htm"
1>Diploma_online - 2 error(s), 0 warning(s)
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Storage Class in C (extern)

stdio.h | file1.c | main.c

```c
/*
// * main.c
// *
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"
  extern int  yyy ;

int  main ()
{
yyy = 1 ;

    return 0 ;
}
```

WHAT IS THE OUTPUT ? WHY?

Storage Class in C (extern)

stdio.h | file1.c | main.c

```c
/*
// * main.c
// *
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"
  extern int  yyy ;

int  main ()
{
yyy = 1 ;

    return 0 ;
}
```


```
:40:48 **** Incremental Build of configuration Debug for project C_Programming ****
fo: Internal Builder is used for build
c -ansi -O0 -g3 -Wall -c -fmessage-length=0 -o main.o "..\\main.c"
c -ansi -o C_Programming.exe file1.o main.o
in.o: In function `main':
\courses\new_diploma\Diploma online\Eclipse_ws\C_Programming\Debug/../main.c:13: undefined reference to `yyy'
llect2: ld returned 1 exit status

:40:49 Build Failed. 1 errors, 0 warnings. (took 469ms)
```

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Storage Class in C (extern)

stdio.h | file1.c | main.c

```c
/*
// * main.c
// *
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"
  extern int  yyy ;

int  main ()
{
yyy = 1 ;
printf ("%d",yyy);
    return 0 ;
}
```

THINK IN DEPTH ☺

HOW TO PASS THIS ERROR

WITHOUT REMOVE EXTERN

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Storage Class in C (extern)

stdio.h | file1.c | main.c

```c
/*
// * main.c
// *
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"
  extern int  yyy ;

int  main ()
{
yyy = 1 ;
printf ("%d",yyy);
    return 0 ;
}
int  yyy ;
```

Problems | Tasks | Console | Properties

<terminated> (exit value: 0) C_Programming.exe [C/C++ Application] D:\courses\new_diploma\Diploma online\Eclipse_ws\C_Programmin

```
1
```

Storage Class in C (register)

- The **variables** defined as **the register is allocated the memory into the CPU registers** depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables,
  - i.e., we can **not use &operator** for the register variable.
- The **access time** of the register variables is faster than the automatic variables.
- The initial default value of the **register local variables is 0**.
- The r**egister keyword** is used for the variable which should be stored in the CPU register. However, ***it is compiler?s choice whether or not; the variables can be stored in the register***.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- **Static variables** can not be stored into the register since **we can not use more than one storage specifier for the same variable**.

Storage Class in C (register)

Example 1

```c
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
printf("%d",a);
}
```

Output:

```
0
```


Storage Class in C (register)

```c
#include <stdio.h>
int main()
{
register int a = 1; // vari
printf("%d",&a);
}
```

WHAT IS THE OUTPUT ? WHY?

Storage Class in C (register)

```c
#include <stdio.h>
int main()
{
register int a = 1; // varia
printf("%d",&a);
}
```

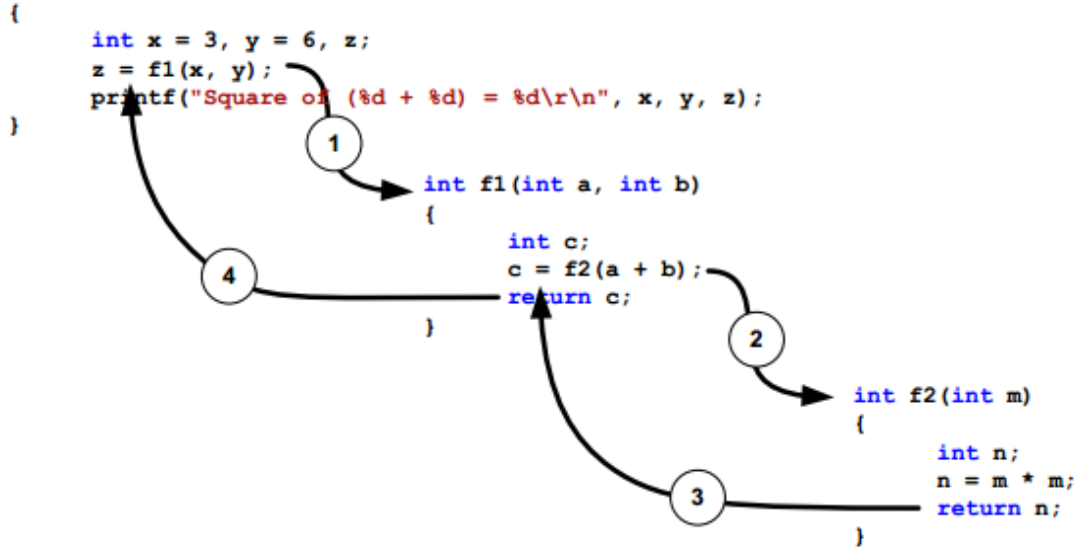
We can not dereference the register variables,
i.e., we can **not use &operator** for the register variable.

```
Output
Show output from: Build
1>------ Build started: Project: Diploma_online, Configuration: Debug Win32 ------
1>Compiling...
1>main.c
1>c:\users\public\documents\vector canape 15\extras\diploma_online\main.c(7) : error C2103: '&' on register variable
1>Build log was saved at "file://c:\Users\Public\Documents\Vector CANape 15\Extras\Diploma_online\Debug\BuildLog.htm"
1>Diploma_online - 1 error(s), 0 warning(s)
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

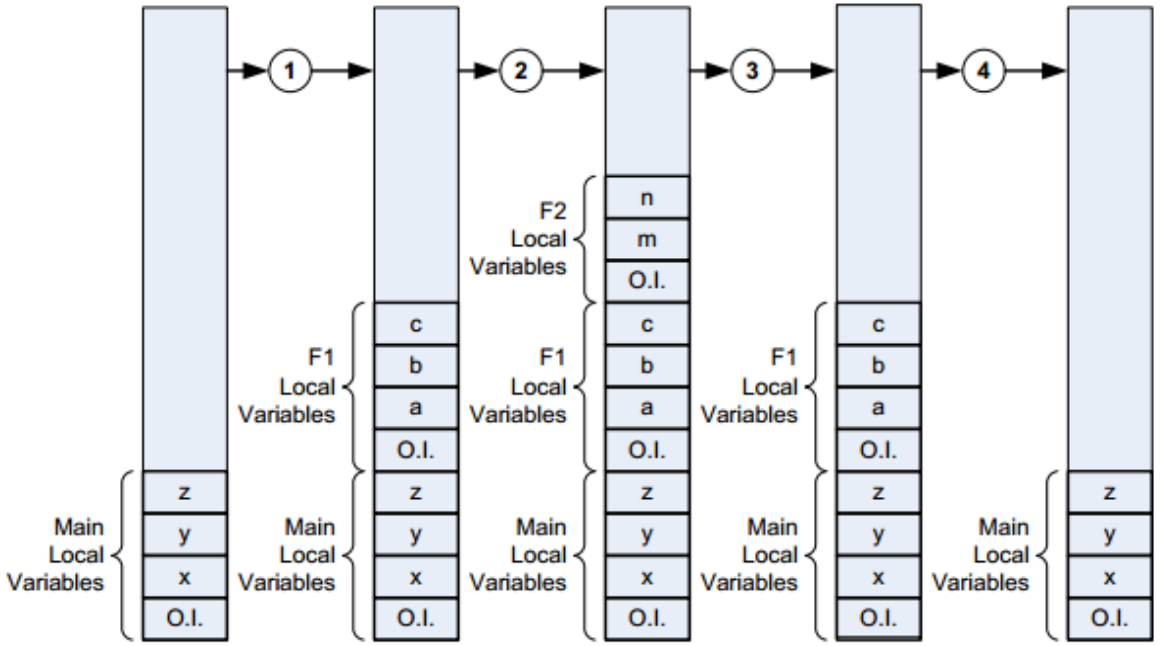
Class	Scoop	Lifetime	Storage	Note
auto	Block	Block	Stack	✓variables are always local **(automatic)** and are stored on the stack. ✓we don't need to write it, because all the variables default is auto.
register	Block	Block	CPU register	✓keep the variable in a register if at all possible. Otherwise it is stored on the stack. ✓We cannot get the address of such variable.
static	Local Global	Program	Data Segment	it Visible only within the scope of the variable: ✓using static variable to reduce the coupling between modules ✓ the value of the static variable will not be destroyed even if the function run ended, and still use this value in memory.
extern	All files	Program	Data Segment	✓Meaning "foreign" … ✓tell the compiler: there is this variable, it may not exist in the current file, but it will certainly be present in one source file in the project. ✓most commonly used state when the project is more than one file, ✓call the total of variables and functions of another file. ✓**don't create another instance of it or there will be a name collision at link time.**

Calling Mechanism

```
void main()
{
        int x = 3, y = 6, z;
        z = f1(x, y);
        printf("Square of (%d + %d) = %d\r\n", x, y, z);
}

int f1(int a, int b)
{
        int c;
        c = f2(a + b);
        return c;
}

int f2(int m)
{
        int n;
        n = m * m;
        return n;
}
```

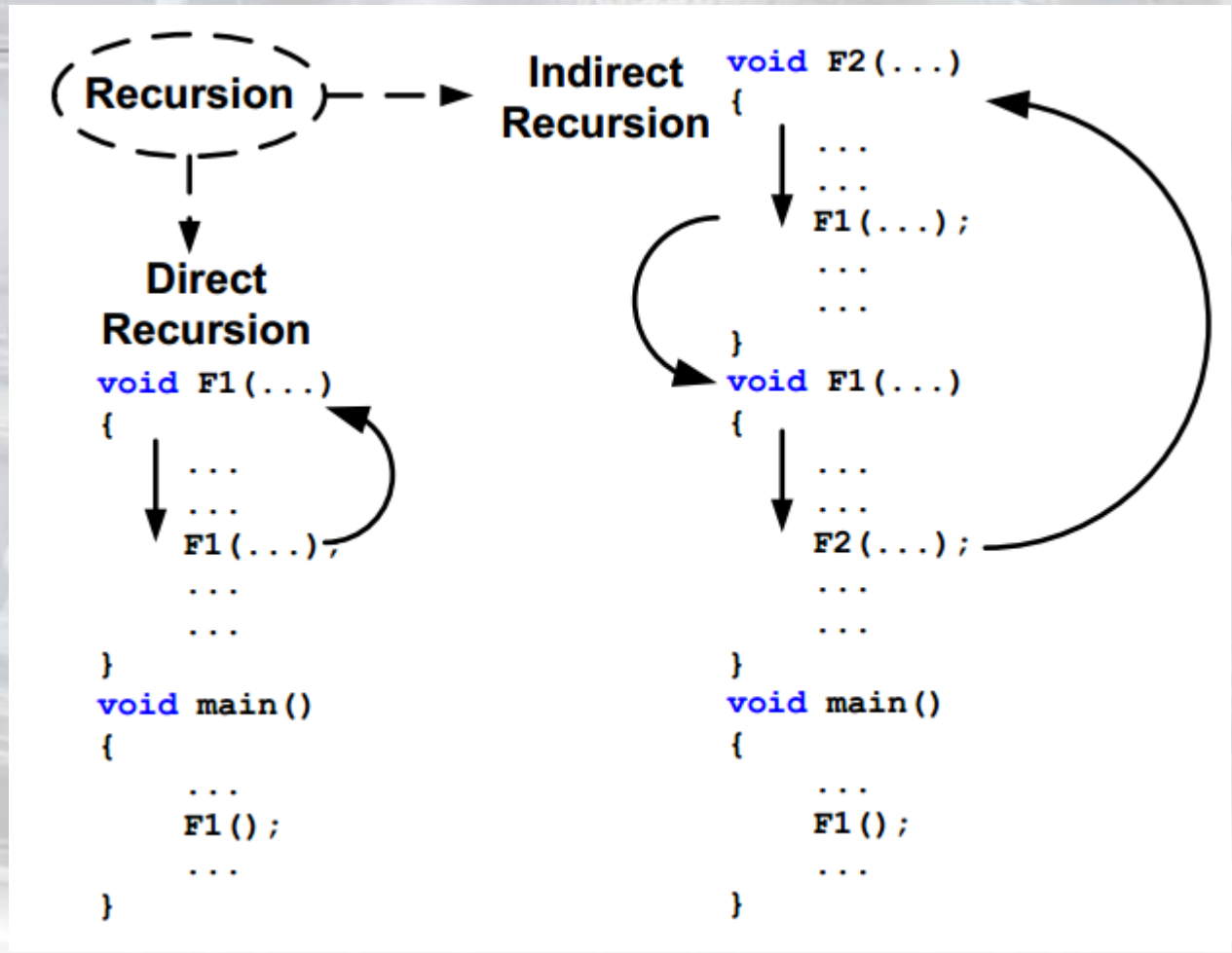


O.I. : Other Information like … caller address



Recursion

- Recursion is a situation happens when a function calls itself directly or indirectly.



Recursion

- Is recursion useful?
  Recursion is an alternative way to repeat the function execution with the same or variant parameters values. In another way recursion is an indirect way to make a loop; simply you can convert any recursion to a normal loop

Infinite Printing Loop

Normal Loop

```
#include "stdio.h"

void printhello()
{
      printf("hello\r\n");
}

void main()
{
      while(1)
            printhello();
}
```

Recursive Loop

```
#include "stdio.h"

void printhello()
{
      printf("hello\r\n");
      printhello();
}

void main()
{
      printhello();
}
```

Recursion

- Is recursion useful?
  Recursion is an alternative way to repeat the function execution with the same or variant parameters values. In another way recursion is an indirect way to make a loop; simply you can convert any recursion to a normal loop

Finite Printing Loop

Finite Normal Loop

```
#include "stdio.h"

void printhello()
{
      printf("hello\r\n");
}

void main()
{
      int i = 0;
      while((i++)<10)
            printhello();
}
```

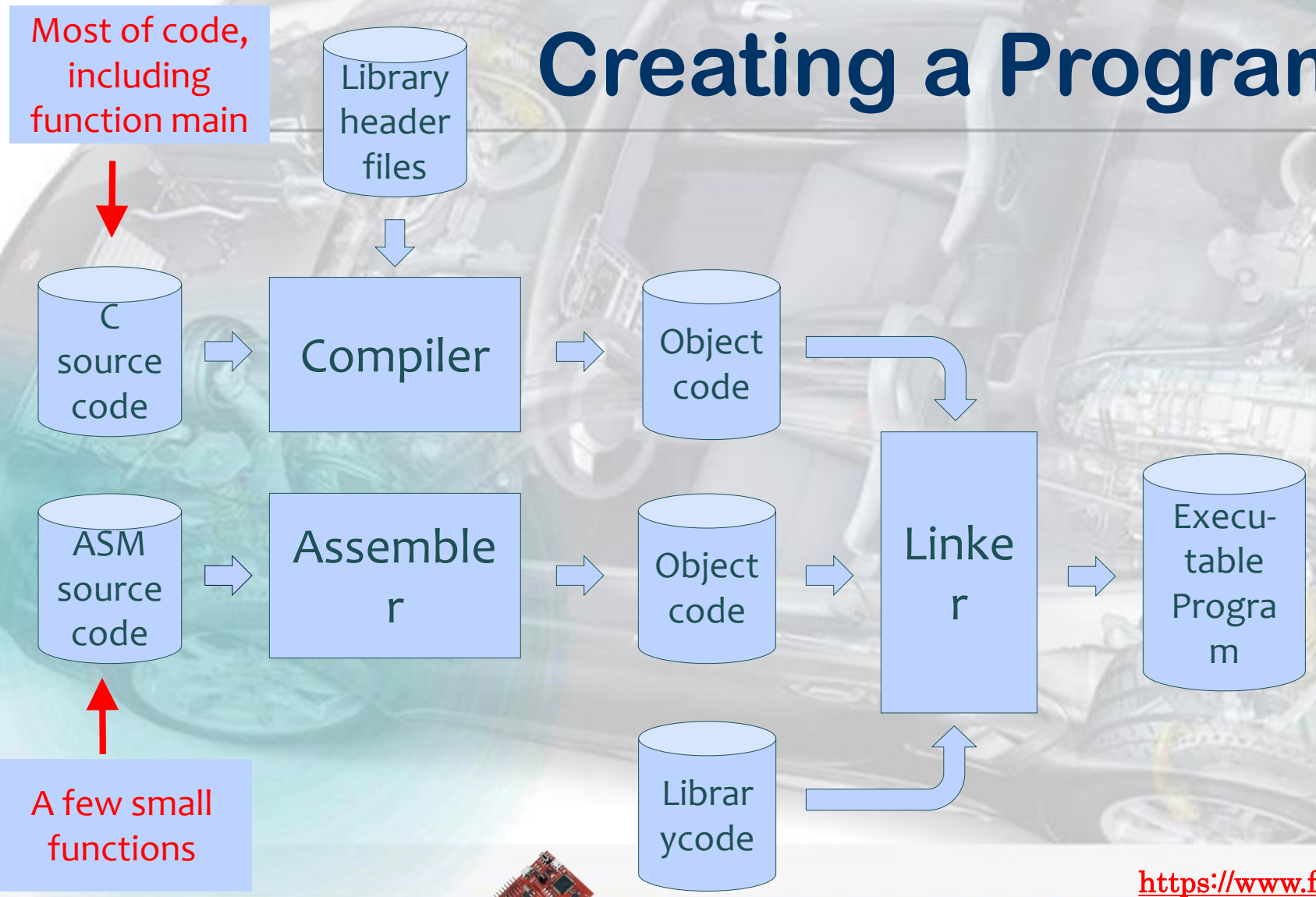
Finite Recursive Loop

```
#include "stdio.h"

void printhello(int
downCounter)
{
      printf("hello\r\n");
      downCounter--;
      if(downCounter>0)

      printhello(downCounter);
}

void main()
{
      printhello(10);
}
```

Functions in Depth (ARM "CASE Study")

increasing **focus** and **concentration**

Creating a Program

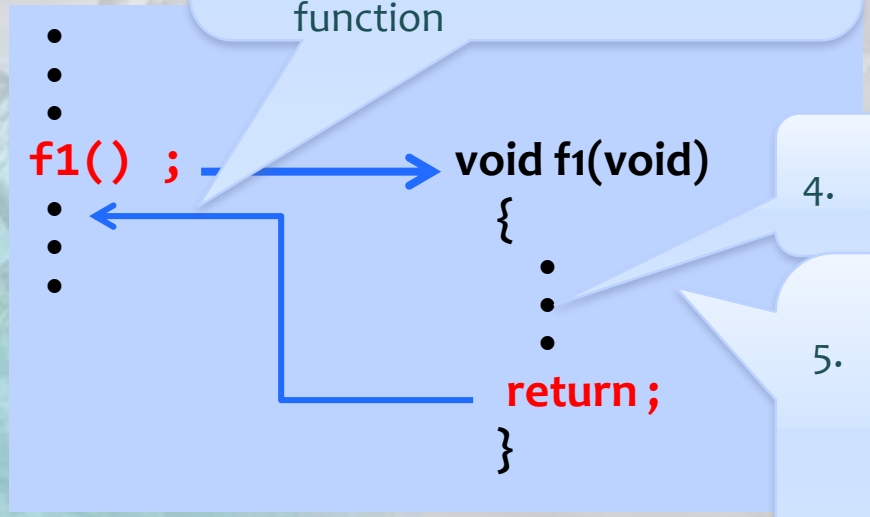


FUNCTION CALL AND RETURN

Simple Call-Return in C

1. Suspend the current sequence
2. Record the return address
3. Transfer control to the function

Branch with Link Instruction (BL)



4. Execute the function

5. Use the recorded return address to go back and resume the suspended code.

Branch Indirect Instruction (BX)

FUNCTION CALL AND RETURN

ARM instructions used to call and return from functions

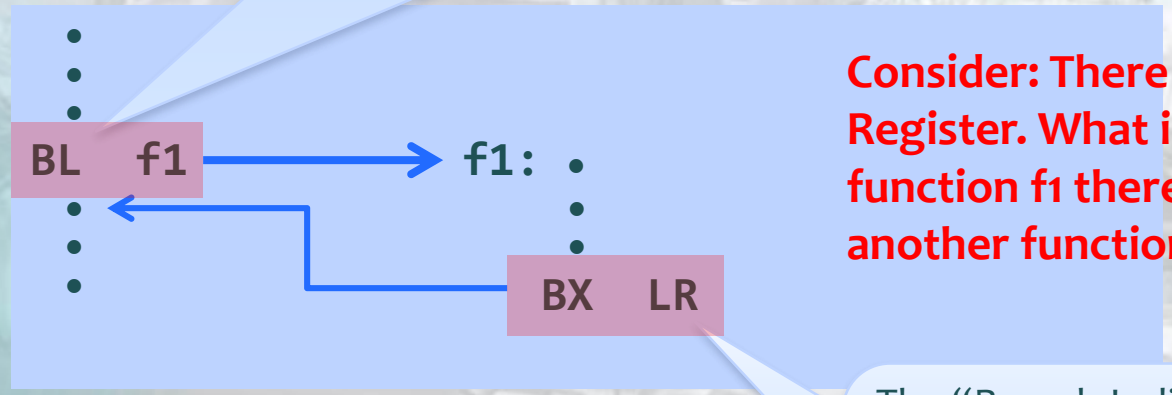
Instruction	*Format*	*Operation*
Branch with Link	BL *label*	***Function Call:*** LR ← return address, PC ← address of label
Branch Indirect	BX LR	***Function Return:*** PC ← LR

LR (aka R14) and PC (aka R15) are two of the 16 registers inside the CPU.
LR is the "Link Register" – used w/function calls to hold the return address
PC is the "Program Counter" – holds the address of the next instruction.

FUNCTION CALL AND RETURN

Simple Call-Return in Assembly

The "Branch with Link" instruction (BL) saves the the address of the instruction immediately following it (the return address) in the Link Register (LR).

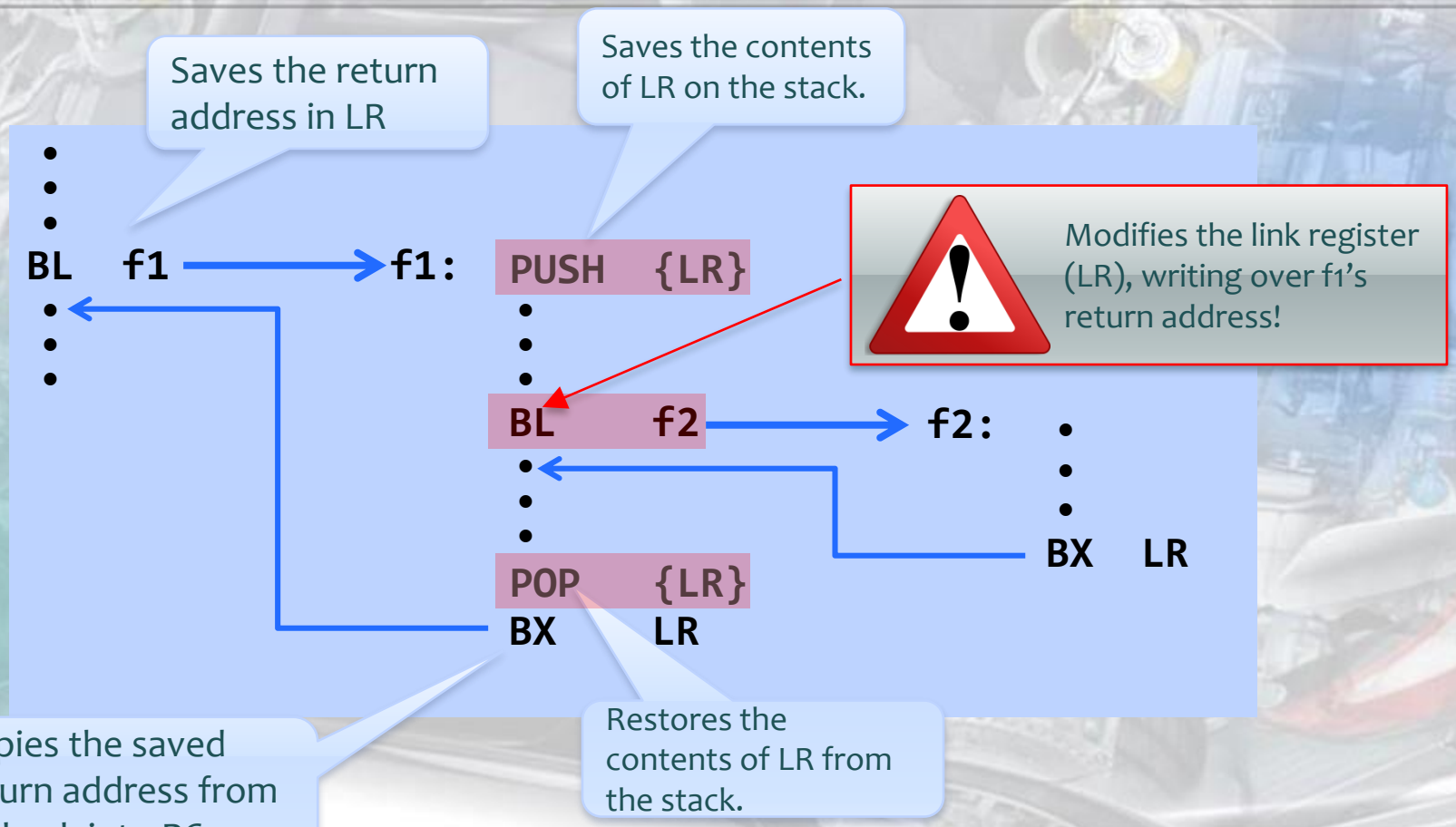


Consider: There is only one Link Register. What if inside function f1 there is a call to another function?

The "Branch Indirect" instruction (BX) copies the return address from LR into PC, thus transferring control back to where the function had been called.

FUNCTION CALL AND RETURN

Nested Call-Return in Assembly



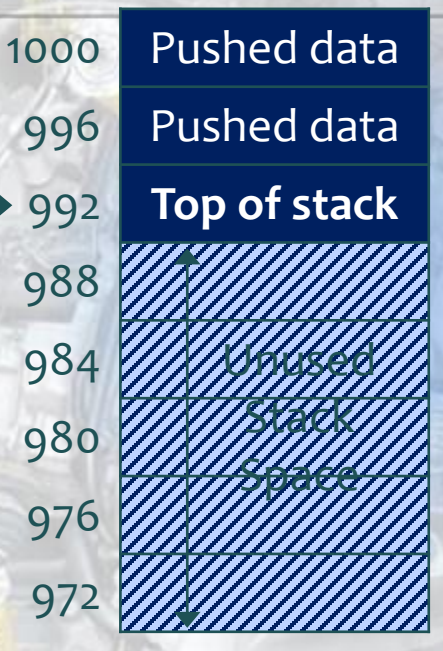
FUNCTION CALL AND RETURN

ARM instructions used in functions that ***call other functions***

SP is one of the 16 CPU registers (R13) – holds the address of the top of the stack.

(register R13)

SP	992



Instruction	*Format*	*Operation*
Push registers onto stack	PUSH *register list*	SP ← SP – 4 × #registers Copy registers to mem[SP]
Pop registers from stack	POP *register list*	Copy mem[SP] to registers, SP ← SP + 4 × #registers

"register list" format: { reg, reg, reg-reg, … }

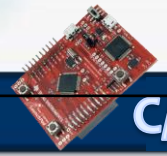
ACCESSING PARAMETERS INSIDE THE FUNCTION

How parameters are passed to a function by the compiler

C Function Call	Compiler Output
`int8_t x8 ;`	`LDRSB R0,x8 // R0 <-- x8`
`int32_t y32 ;`	`LDR R1,y32 // R1 <-- y32`
`int64_t z64 ;`	`LDRD R2,R3,z64 // R3.R2 <-- z64`
`foo(x8, y32, z64) ;`	`BL foo`

`foo(5, -10, 20) ;`	`MOV R0,5 // R0 <-- 5`
	`MOV R1,-10 // R1 <-- -10`
	`MOV R2,20 // R3.R2 <-- 20`
	`MOV R3,0`
	`BL foo`

Inside foo, you must use the register copies of the actual arguments.



INLINE ASSEMBLY

increasing **focus** and **concentration**

- **The inline assembler** lets us embed assembly-language instructions in our C and C++ source programs without extra assembly and link steps.
- **Inline assembly** code can use any C or C++ variable or function name that is in scope.

Using GCC

```
__asm__("movl %edx, %eax\n\t" "addl $2, %eax\n\t");
```

Using VC++

```
__asm { mov eax, edx add eax, 2 }
```

```
asm {
    mov ax,a
    mov bx,b
    add ax,bx
  mov c,ax
}
```

example

```
 /* Add 10 and 20 and store result into
register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "addl %ebx, %eax;"

    );
```

```
1
2  int xxx=1 ;
3
4  void file1_func1(int x )
5  {
6      x += 3 ;
7      printf ("file1_func1 x=%d \n",x);
8
9      /* Add 10 and 20 and store result into register %eax
10     __asm__ ( "movl $10, %eax;"
11               "movl $20, %ebx;"
12               "addl %ebx, %eax;"
13     );
14 }
```

```
10          __asm__ ( "movl $10, %eax;"
0040140d:     mov     $0xa,%eax
00401412:     mov     $0x14,%ebx
00401417:     add     %ebx,%eax
14          }
00401419:     leave
0040141a:     ret
0040141b:     nop
0040141c:     nop
0040141d:     nop
0040141e:     nop
0040141f:     nop
            main:
00401420:     lea     0x4(%esp),%ecx
```


INLINE Function

increasing **focus** and **concentration**

Inline function

- **Inline Function** are those function whose definitions are small and be substituted at the place where its function call is happened. Function substitution is totally compiler choice

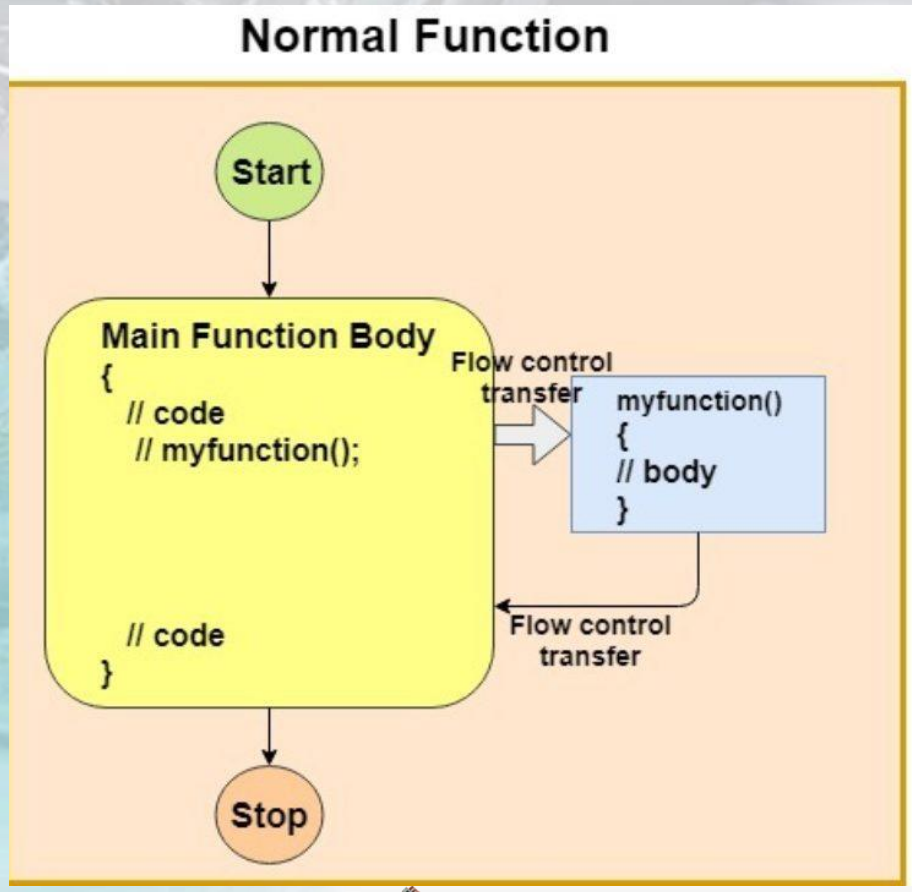
c99

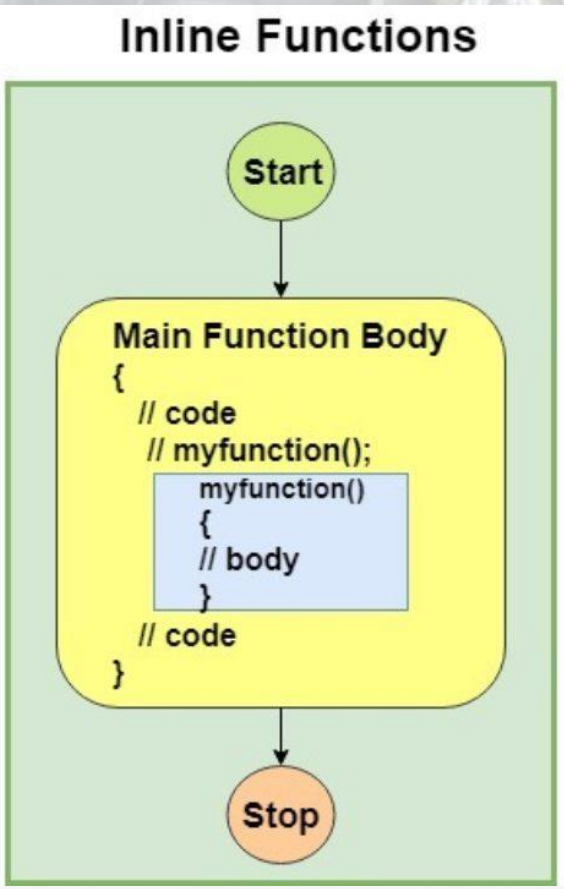
```c
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"

void file1_func1(int x);

/* Inline function in C */
inline int example(int x)
{
    x += 3 ;
    return x;
}

int  main ()
{
    int y = 7 ;
    y=example (y);
    printf ("%d",y);
    return 0 ;
}
```

ansi C

```c
// *  Created on: Aug 20, 2020
// *      Author: Keroles Shenouda
 */

#include "stdio.h"

void file1_func1(int x);

/* Inline function in C */
__inline int example(int x)
{
    x += 3 ;
    return x;
}

int  main ()
{
    int y = 7 ;
    y=example (y);
    printf ("%d",y);
    return 0 ;
}
```

Inline function

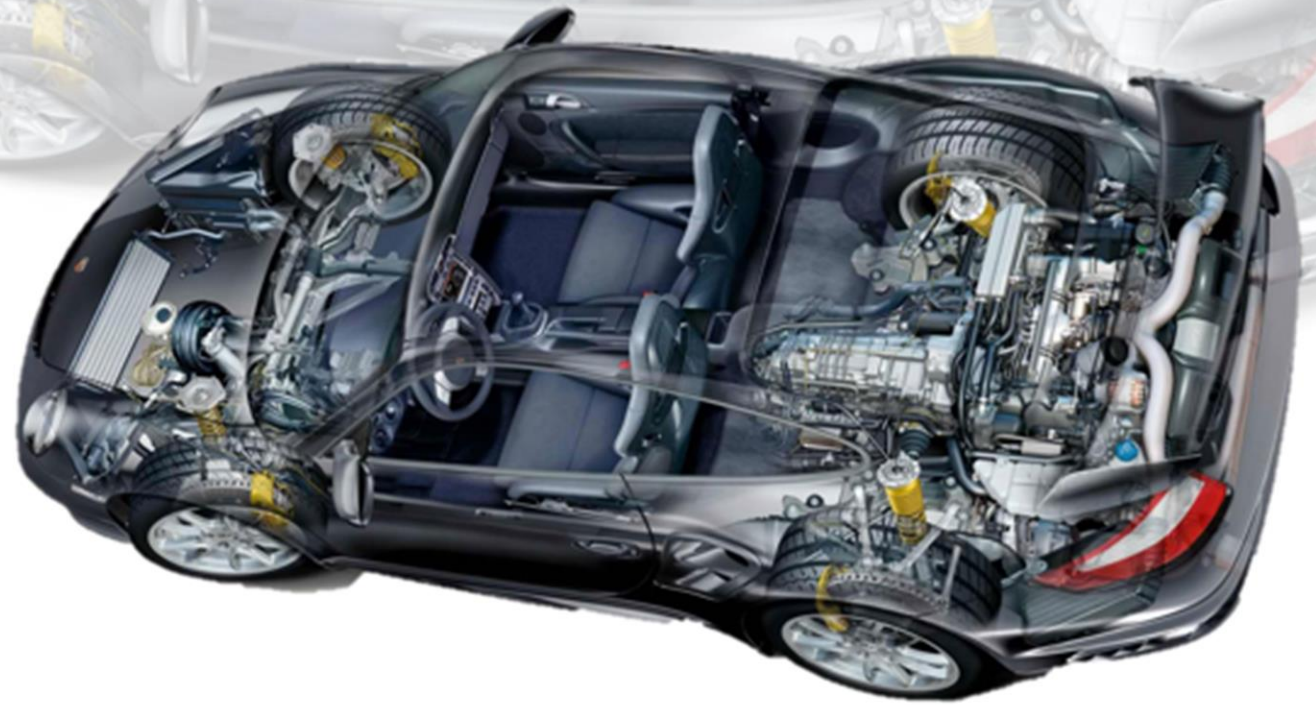




#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Thanks and Good Luck



ENG. Keroles Shenouda
https://www.facebook.com/groups/embedded.system.KS/

https://www.facebook.com/groups/embedded.system.KS/