#LEARN_IN DEPTH

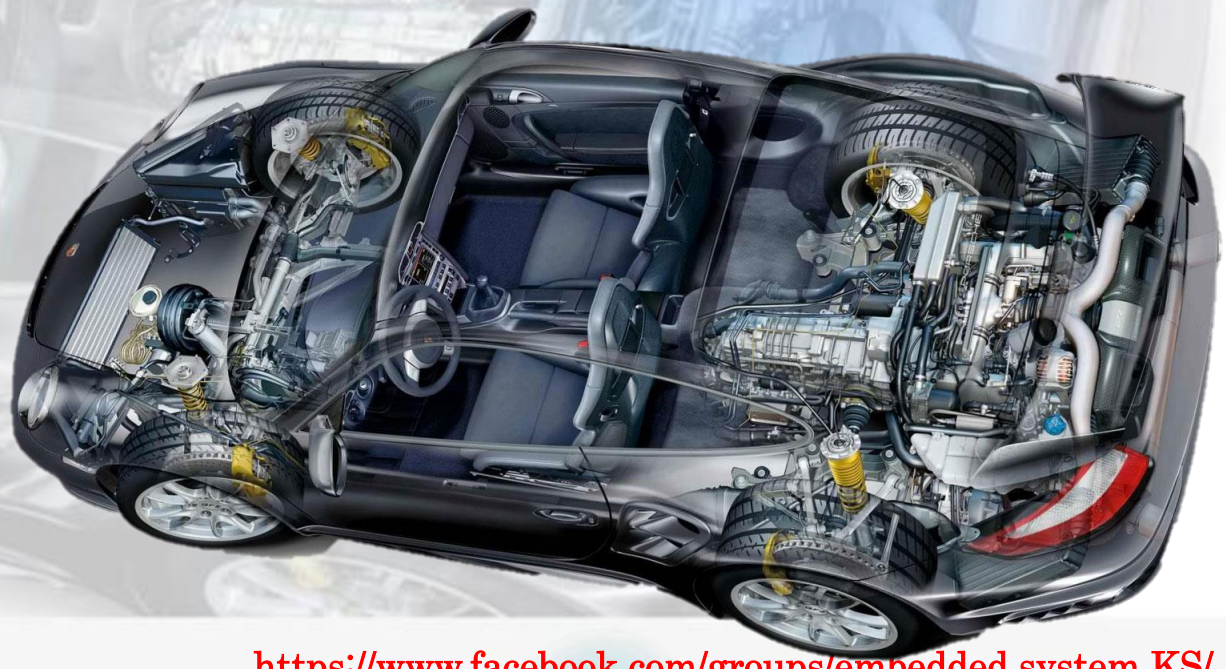
#Be_professional_in_embedded_system

Unit2 Lesson 8
C Pointers

- Pointer Types
- Pointer tricks

Learn in Depth

ENG.KEROLES SHENOUDA

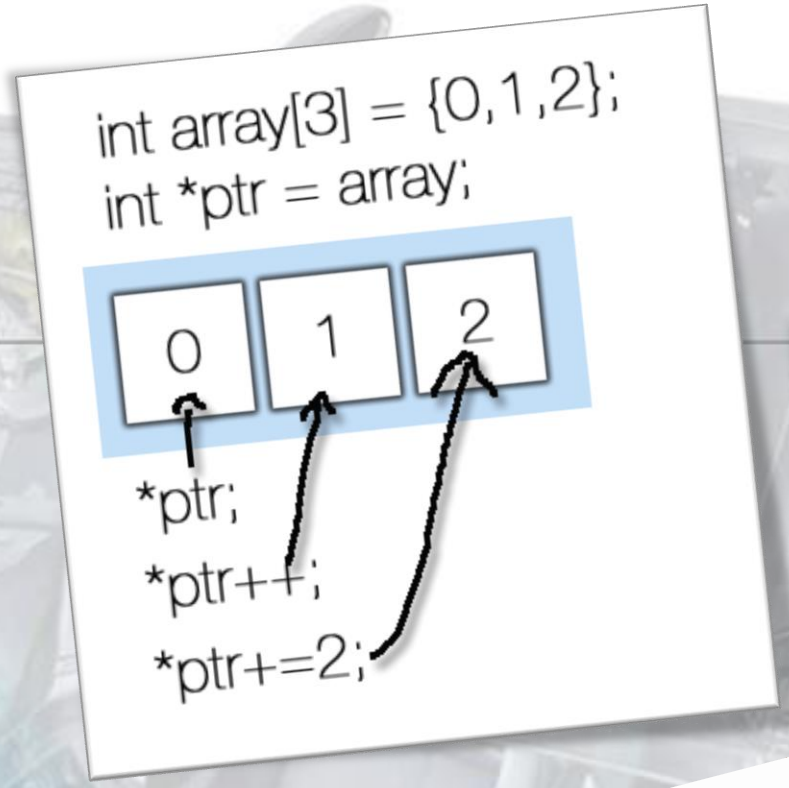


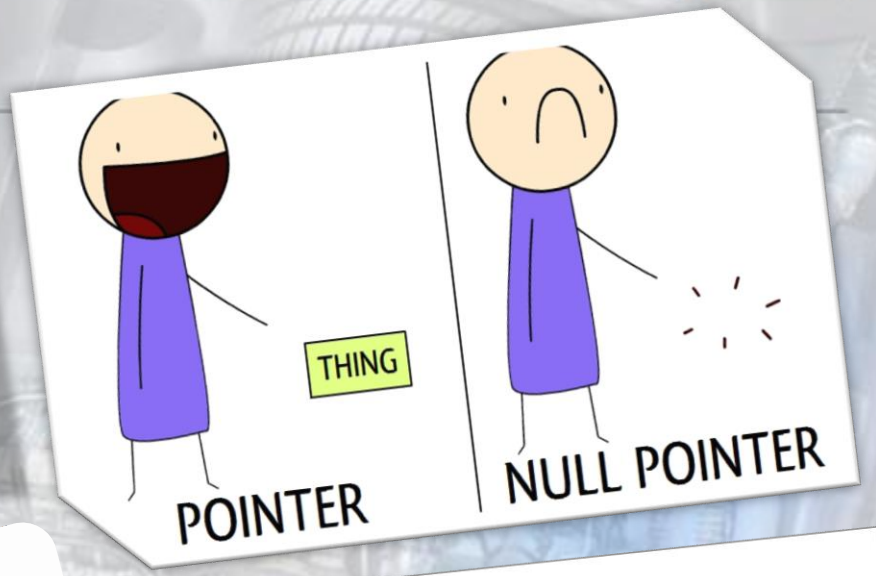
https://www.facebook.com/groups/embedded.system.KS/

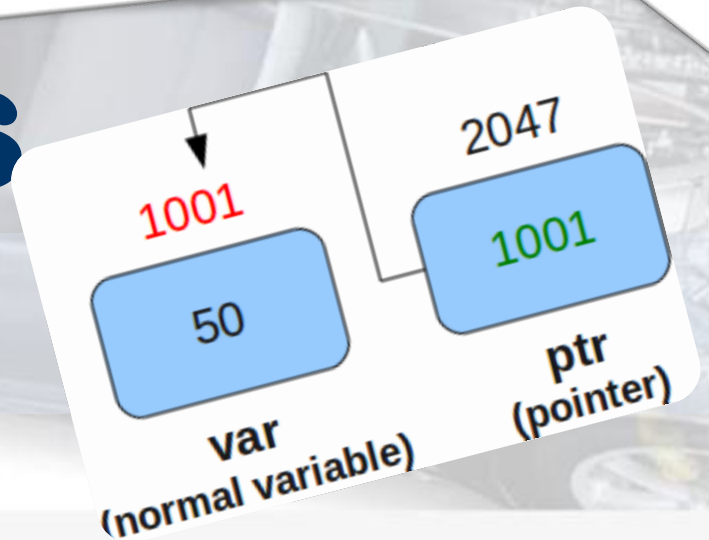
#LEARN_IN DEPTH

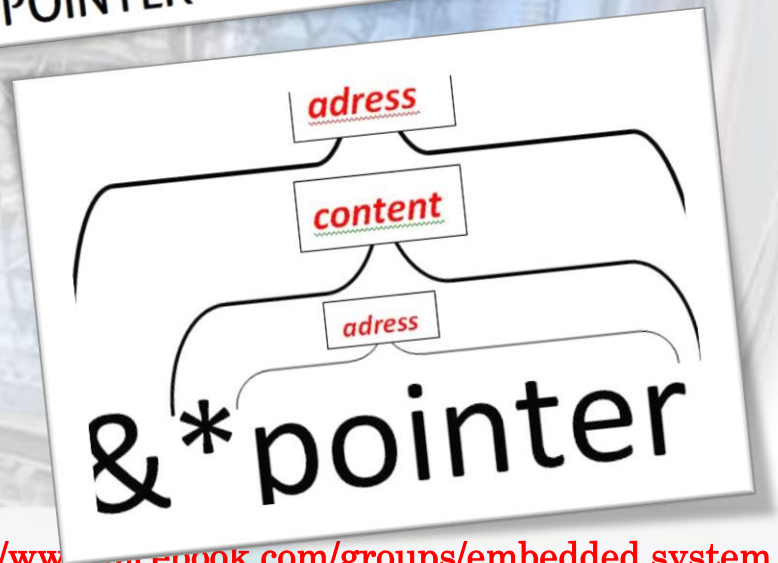
#Be_professional_in_embedded_system

Pointers





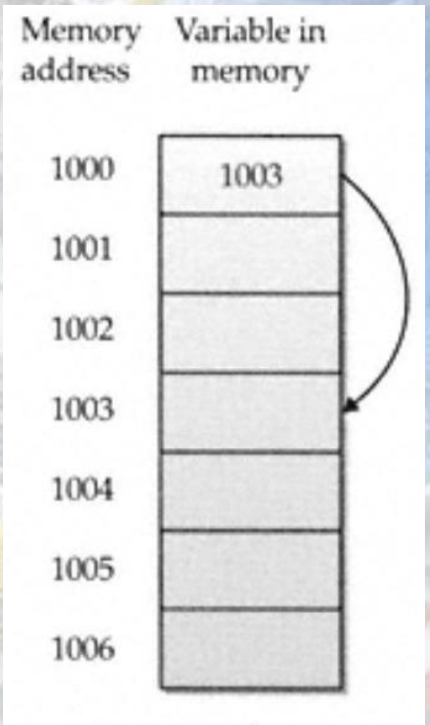




https://www.facebook.com/groups/embedded.system.KS/

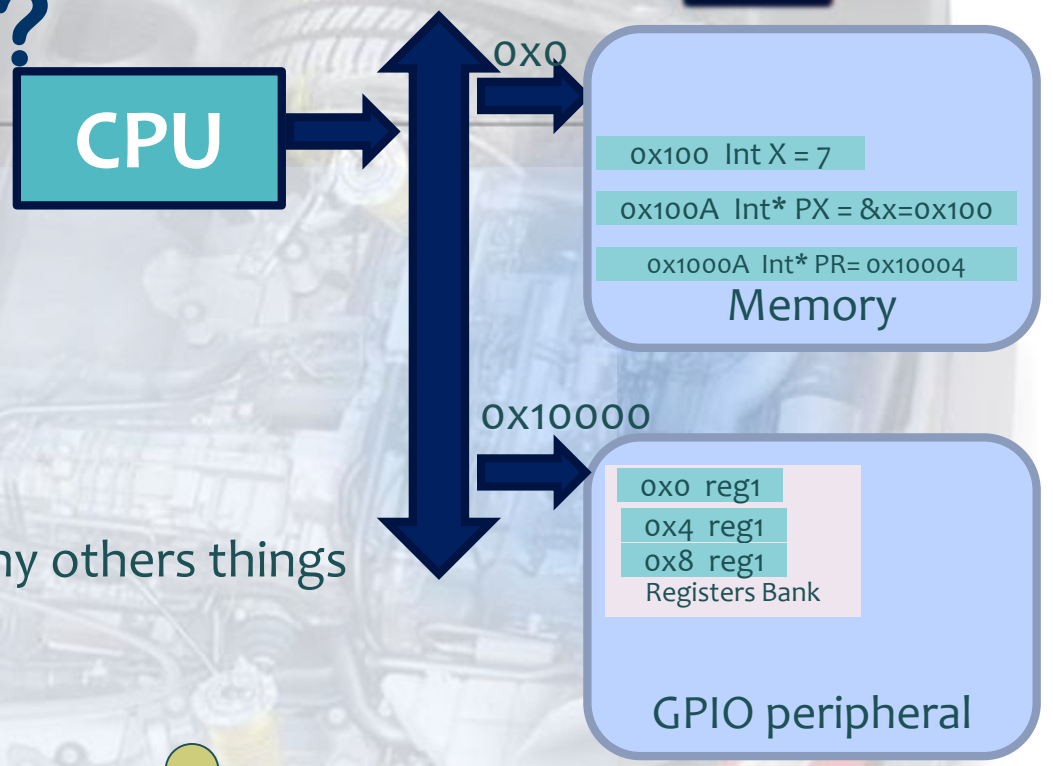
What Are Pointers?

- A ***pointer*** is a **variable** **that holds a memory address**.
- This address is the location of another object (**typically another variable**) in memory.
  For example, if one variable contains the address of another variable, ***the first variable*** is **said** ***to point to*** **the second**.



Why do we need Pointer?

- Simply because it's there!
- Pointers make C language more powerful
  - Configure the peripheral register addresses
  - Read/Write into peripheral data registers
- Read/write into SRAM/FLASH locations and for many others things



Remember this?

```
scanf("%d", &i);
```

Pointer Size

- Pointers generally have a fixed size, It depends upon different issues like Operating system, CPU architecture.
- The Pointer Size is **Equivalent** to **memory Location address Size**
  - **Ex: For 64 machine the size = 8 Bytes (64bit)**
  - **For 32 Machine the size = 4byte = 32bit**
- It is common to all data types like int *, float * etc.
-

<Pointer data Type> *<Variable_Name>
<Pointer data Type>* <Variable_Name>

- char*
- short int*
- int*
- long int*
- long long int*
- unsigned char*
- unsigned short int*
- unsigned int*
- unsigned long int*
- unsigned long long int*

```
..\pointers.c:19: warning: initialization makes pointer from integer without a cast
```


Pointer casting

pointers.c

```c
 *
 *  Created on: Oct 15, 2020
 *      Author: Keroles Shenouda
 */

#include "stdio.h"

int main ()
{
    //normal number
    long long int rand_address = 0x00000000FFFFAAAA ;
    printf ("rand_address = %llx \n",rand_address);

    //0x00000000FFFFAAAA :for the compiler point of view it is still considered long long number not address
    //char* for the compiler it is a pointer
    //So the compiler will see type mismatch
    char* Paddress1 = 0x00000000FFFFAAAA ;
    printf ("Paddress1 = %llx \n",Paddress1);

    //So we need to tell the compiler it is address not number by the casting
    char* Paddress2 = (char*)0x00000000FFFFAAAA ;
    printf ("Paddress2 = %llx \n",Paddress2);

    return 0 ;
}
```

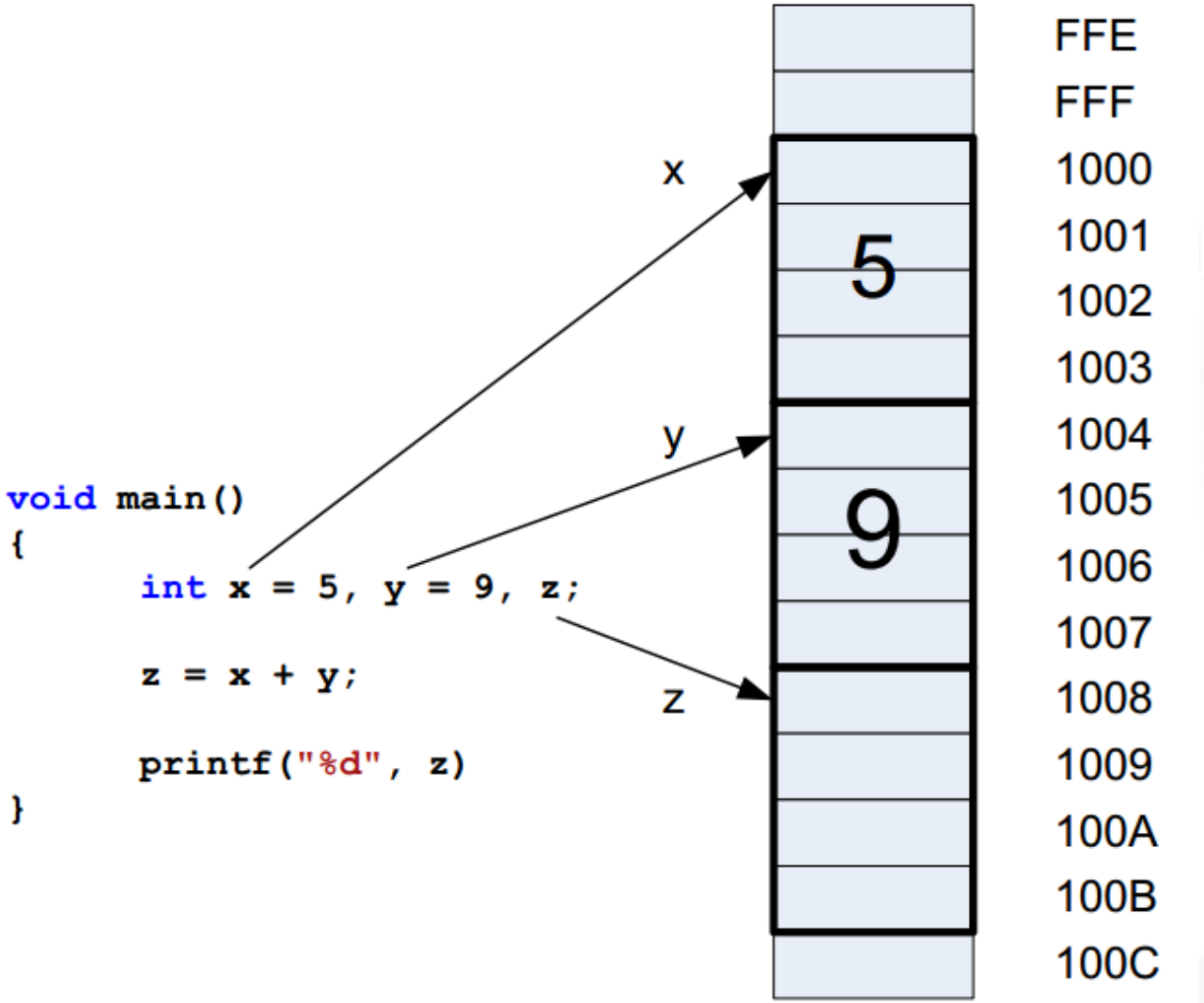
Problems Tasks Console Properties

<terminated> (exit value: 0) pointers_lab.exe [C/C++ Application] D:\courses\new_diploma\Diploma online\Eclipse_ws\pointers_lab\Debug\pointers_lab.exe (10/15/20, 12:13 PM)

```
rand_address = ffffaaaa
Paddress1 = ffffaaaa
Paddress2 = ffffaaaa
```


Pointers
Pointer Variables

- If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is
  *type *name;*



Pointers
Pointer Variables

Problems | AVR Supported MCUs | Tasks | Console | Properties

```
<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Course\ses
x at location 0x61ff2c contains 5

y at location 0x61ff28 contains 9

z at location 0x61ff24 contains 14
```

main.s | main.c

```c
//Prepared by Eng.Keroles
#include <stdio.h>

int main(int argt ,char**argv) {
    int x = 5, y = 9, z;
    printf("x at location 0x%x contains %d\r\n", &x, x);
    printf("y at location 0x%x contains %d\r\n", &y, y);
    z = x + y;
    printf("z at location 0x%x contains %d\r\n", &z, z);
    return 0   ;
}
```

Pointers
Pointer Variables

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

main.s | main.c

```c
//Prepared by Eng.Keroles
#include <stdio.h>

int main(int argt ,char**argv) {


    int x = 6;
    int* px;
    px = &x;
    printf("%x %x\r\n", &x, px);
    printf("%d %d\r\n", x, *px);

    return 0   ;
}
```

Problems | AVR Supported MCUs | Tasks | Console

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\c

```
61ff28 61ff28


6 6
```

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

main.s | *main.c

```c
//Prepared by Eng.Keroles
#include <stdio.h>
int main(int argt ,char**argv) {
    int x = 5;
    int y = 7;
    int* p;
    p = &x;
    printf("x = %d, value pointed by p = %d\r\n", x, *p);
    *p = 14;
    printf("x = %d, value pointed by p = %d\r\n", x, *p);
    p = &y;
    printf("x = %d, value pointed by p = %d\r\n", x, *p);
    *p = 20;
    printf("y = %d, value pointed by p = %d\r\n", y, *p);
    p = 0;
    //*p = 15; //Wrong and the program will crash here
    return 0   ;
}
```

Pointers
Pointer Variables

Problems | AVR Supported MCUs | Tasks | Console | Properties | AV

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Course\session2\

```
x = 5, value pointed by p = 5

x = 14, value pointed by p = 14

x = 14, value pointed by p = 7

y = 20, value pointed by p = 20
```


What is the Output ?
Unary increment operation for the pointer P++

main.s main.c

```c
//Prepared by Eng.Keroles
#include <stdio.h>
int main(int argt ,char**argv) {
    int* px = 0x0;
    char* py = 0x0 ;
    long long* pz = 0x0 ;
    printf ("px =0x%X , pz=0x%X and py =0x%X \n",px,pz,py);
    px++ ;
    py ++ ;
    pz++ ;
    printf ("px =0x%X , pz=0x%X  and py =0x%X \n",px,pz,py);

    return 0   ;
}
```

Pointers
Pointer Variables

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

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

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Course\session2\Debug\sessio

```
px =0x0 , pz=0x0 and py =0x0
px =0x4 , pz=0x8  and py =0x1
```

Pointer Arithmetic

- There are only **two arithmetic operations** that you can use on pointers:

addition and **subtraction**.

```
1  //Prepared by Eng.Keroles
2  #include <stdio.h>
3  int main(int argt ,char**argv) {
4      int* px = 0x0;
5      char* py = 0x0 ;
6      long long* pz = 0x0 ;
7      printf ("px =0x%X , pz=0x%X and py =0x%X \n",px,pz,py);
8      px++ ;
9      py ++ ;
10     pz++ ;
11     printf ("px =0x%X , pz=0x%X  and py =0x%X \n",px,pz,py);
12
13     return 0   ;
14 }
```

Increment one of(my type)

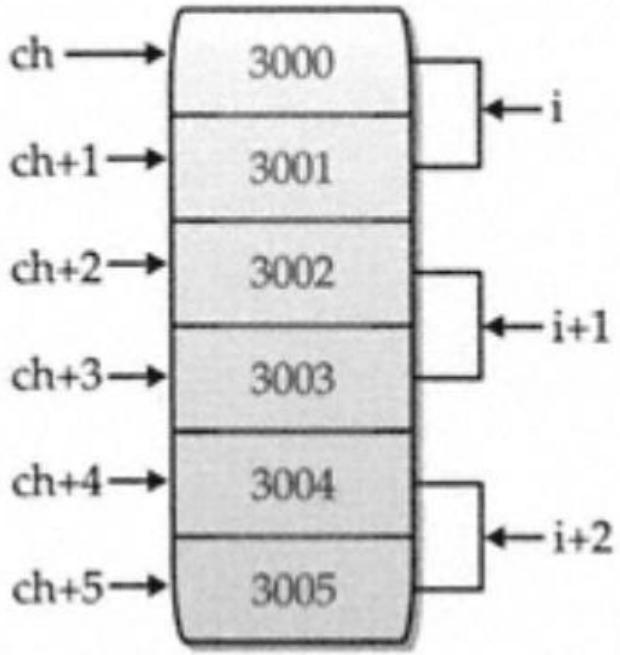
The reason for this is that each time px is incremented, it will point to the next integer

rule govern pointer arithmetic.

Each time a pointer is incremented,
it points to the memory location
of the next element of its base type

All pointer arithmetic is relative to its base type (assume 2-byte integers)

```
char *ch = (char *) 3000;
int *i = (int *) 3000;
```



Pointer Size is depend only on the machine

Note

&PX

4 bytes in 32 machine or 8 bytes in 64 machine

The compiler reserve 8 bytes (for 64 bits machine) of memory for the

definition.

It is worth to remark that the compiler will always reserve 8 bytes independently from the pointer datatype

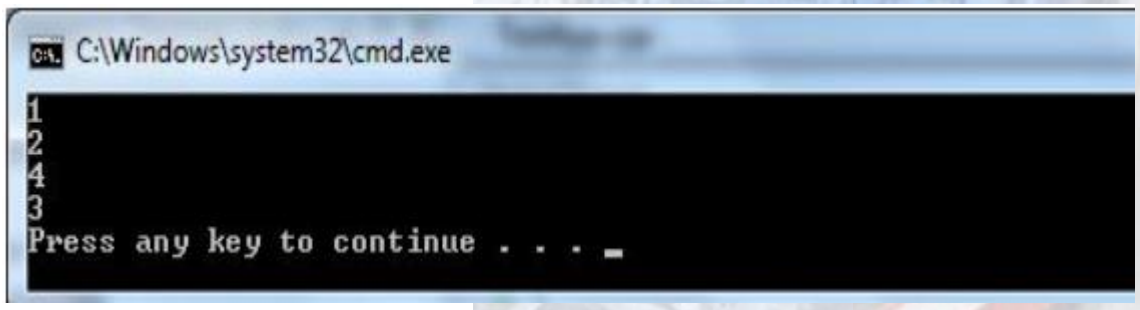
(which could be char*, int* etc etc).

- In other words, the pointer datatype doesn't control the memory size of the pointer variable.
- The pointer datatype **identifies** the **behavior of the operations carried out on the pointer variable**. (Read/Write)

Pointer to Array

```c
#include "stdio.h"

void main()
{
	int x[5] = {1, 2, 3, 4, 5};
	int* p = x;
	printf("%d\r\n", *p);
	p++;
	printf("%d\r\n", *p);
	p = x + 3;
	printf("%d\r\n", *p);
	p--;
	printf("%d\r\n", *p);

	//x++; Wrong, Array Address is Fixed
}
```

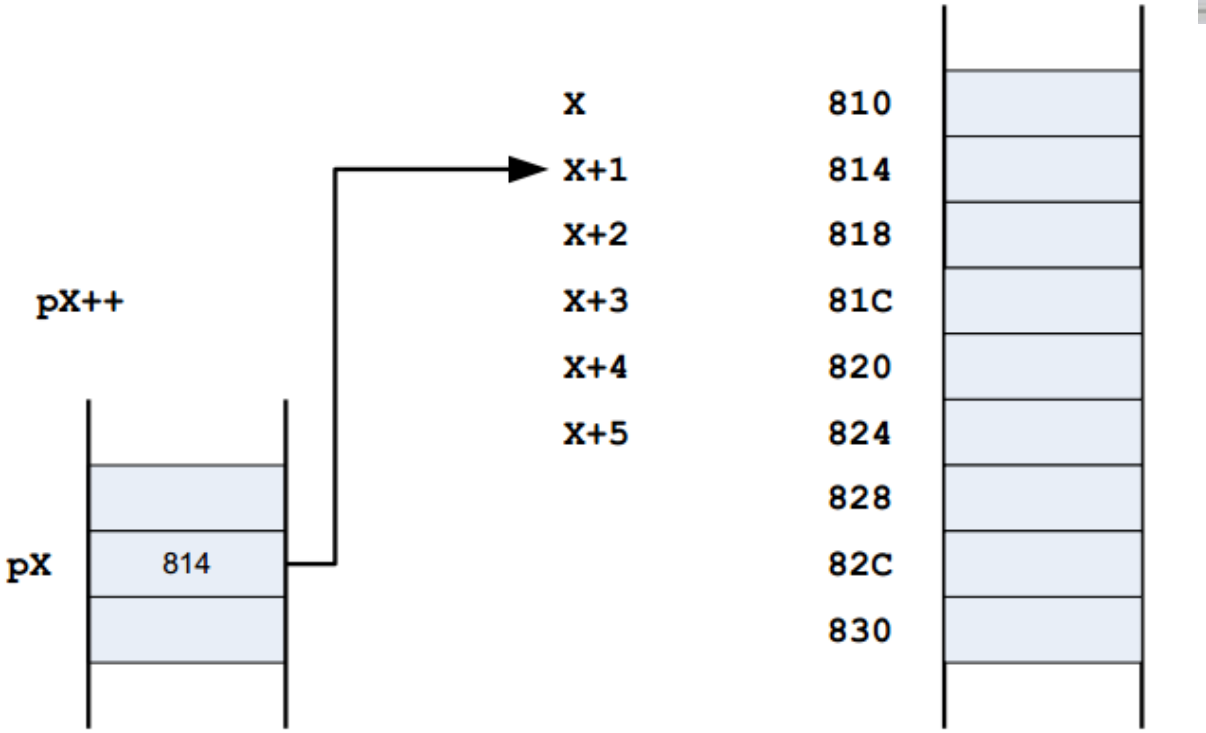

```c
#include "stdio.h"

void main()
{
	int x[5] = {1, 2, 3, 4, 5};
	int* p = x;
	printf("%d\r\n", *p);
	p++;
	printf("%d\r\n", *p);
	p = x + 3;
	printf("%d\r\n", *p);
	p--;
	printf("%d\r\n", *p);

	//x++; Wrong, Array Address is Fixed
}
```

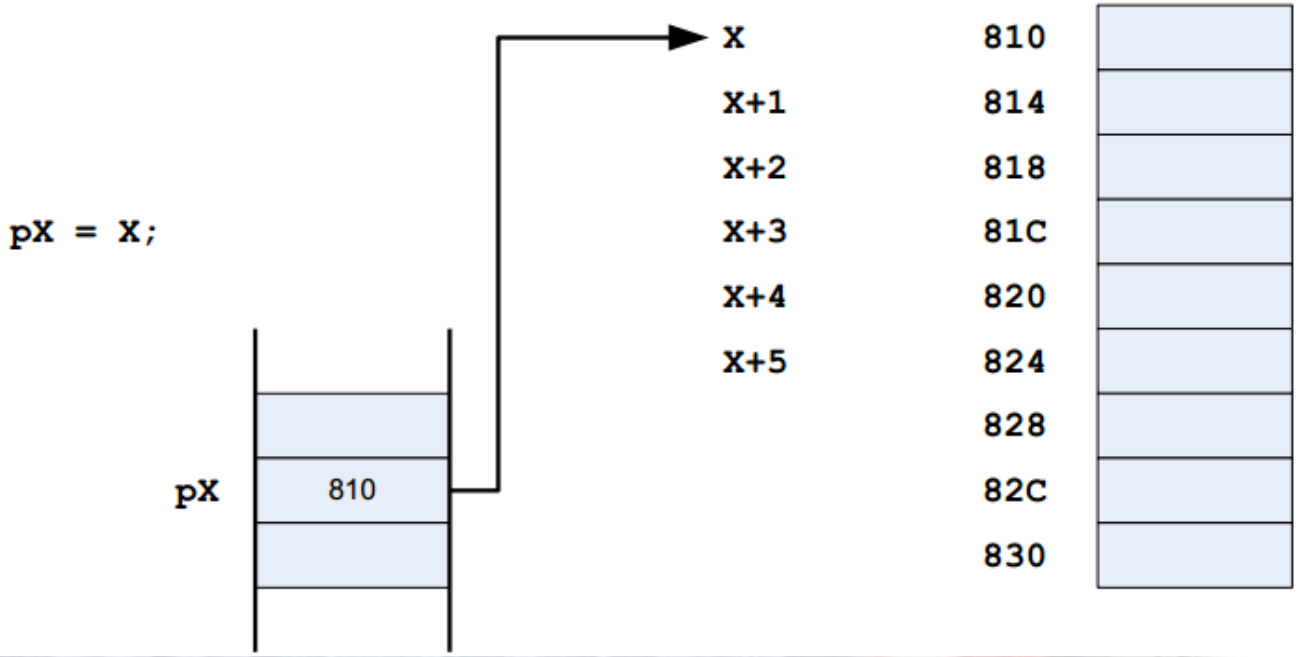


LAB Average of Weights

- It is required to calculate the summation weight of 5 boxes. The user should enter the boxes


```c
#include "stdio.h"

void main()
{
	int X[5];
	int sum = 0;
	int* pX = X;

	for(i=0;i<sizeof(X)/sizeof(int);i++)
		scanf("%d\n", pX + i);

	for(i=0;i<sizeof(X)/sizeof(int);i++)
		printf("%d\n", *pX++);

	pX = X;
	for(i=0;i<sizeof(X)/sizeof(int);i++, pX++)
		sum += *pX;

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


X
X+1
X+2
X+3
X+4
X+5
810
814
818
81C
820
824
828
82C
830
pX++
pX
814


pX = X;
X
X+1
X+2
X+3
X+4
X+5
810
814
818
81C
820
824
828
82C
830
pX
810

Pointers and Arrays

- There is a close relationship between pointers and arrays.
- Consider this program fragment:
  char str[80], *p1;
  p1 = str;
  Here, **p1** has been set to the address of the first array element in **str**.
- To access the fifth element in **str**, you could write

str[4]
or
***(p1+4)**

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Pointer to structure

pointers.c

```
#include "stdio.h"

struct SDataSet
{
    unsigned char data1 ;
    unsigned int data2 ;
    unsigned char data3;
    unsigned short data4 ;
};

struct SDataSet data1 ;
void print_memory_range (char* base , int size)
{
    int i;
    for (i=0;i<size;i++){
        printf ("%p \t %x \n",base, (unsigned char)*base);
        base++ ;
    }

}
int main ()
{
    data1.data1 = 0x11;
    data1.data2 = 0xFFFFEEEE;
    data1.data3 = 0x22;
    data1.data4 = 0xABCD;
    print_memory_range (&data1 , sizeof(data1));

    char* p = &data1 ;
    printf ("p+8 =%x \n",(unsigned char)*(p+8));
    struct SDataSet* PStruct = &data1 ;
    printf ("p+8 =%x \n",(unsigned char) PStruct->data3);

    return 0 ;
}
```

Problems | Tasks | Console | Properties

<terminated> (exit value: 0) pointers_lab.exe [C/C++ Application] D:\courses\new_diploma\

```
00404040        11
00404041        0
00404042        0
00404043        0
00404044        ee
00404045        ee
00404046        ff
00404047        ff
00404048        22
00404049        0
0040404A        cd
0040404B        ab
p+8 =22
p+8 =22
```

Pointer to Structure

```c
//Prepared by Eng.Keroles
#include <stdio.h>
struct SPerson
{
    char m_Name[18];
    int m_ID;
    char m_Age;
    short m_Salary;
    double m_Weight;
};
```

->

```c
int main(int argt ,char**argv) {
    struct SPerson manager =
    {"Mohamed Hady", 162, 39, 3000, 79.5};
    struct SPerson emploees[] = {
            {"Mostafa Said", 163, 30, 1500, 81.0},
            {"Ahmed Salah", 164, 25, 1200, 91.0},
            {"Safa Fayez", 165, 28, 1400, 65.0}};
    int i;
    struct SPerson* p;
    p = &manager;
    printf("manager: %s, %d, %d, %d, %lf\r\n",
            p->m_Name, p->m_ID, (int)p->m_Age,
            (int)p->m_Salary, p->m_Weight);
    p->m_Salary = 4000;
    printf("manager: %s, %d, %d, %d, %lf\r\n",
            manager.m_Name, manager.m_ID,
            (int) manager.m_Age, (int) manager.m_Salary,
            manager.m_Weight);
    p = emploees;
    for(i=0;i<sizeof(emploees)/sizeof(struct SPerson);
            i++, p++)
    {
        printf("emploee %d: %s, %d, %d, %d, %lf\r\n",
                i+1, p->m_Name, p->m_ID,
                (int)p->m_Age, (int)p->m_Salary,
                p->m_Weight);
    }
    return 0   ;
}
```


Pointer to Structure

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

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Course\session2\Debug\session2.exe (5/5/1

```
manager: Mohamed Hady, 162, 39, 3000, 79.500000

manager: Mohamed Hady, 162, 39, 4000, 79.500000

emploee 1: Mostafa Said, 163, 30, 1500, 81.000000

emploee 2: Ahmed Salah, 164, 25, 1200, 91.000000

emploee 3: Safa Fayez, 165, 28, 1400, 65.000000
```

Pointers AND Functions

- Pointers are used efficiently with functions. Using pointers provides two main features:
  - Fast data transfer, because only pointers are transferred.
  - Pointers allow the definition of several outputs for the same function.

Fast Data Transfer Using Pointers

```c
#include "stdio.h"

struct SDate
{
	int m_Day;
	int m_Month;
	int m_Year;
};

struct SStudent
{
	char m_Name[256];
	char m_Description[8192];
	struct SDate m_BirthDate;
	double m_Degrees[10];
	double m_TotalDegrees;
};

struct SStudent SlowUpdateTotalDegree(struct SStudent student)

{
	student.m_TotalDegrees = 0;
	int i = 0;
	for(i=0;i<10;i++)
		student.m_TotalDegrees += student.m_Degrees[i];

	return student;
}
```

Fast Data Transfer Using Pointers

```
void FastUpdateTotalDegree(struct SStudent* pStudent)
{
	pStudent->m_TotalDegrees = 0;
	int i = 0;
	for(i=0;i<10;i++)
		pStudent->m_TotalDegrees += pStudent->m_Degrees[i];
}

void main()
{
	struct SStudent S = {"Ahmed Said", "Ahmeds'description",
			{22, 12, 1990},
			{88, 98, 88, 92, 98, 87, 66, 94, 87, 99}};

	//Method 1: Without Pointers
	S = SlowUpdateTotalDegree(S);
	printf("Total degrees of %s is %2.2lf (%2.2lf%%)\r\n",
		S.m_Name, S.m_TotalDegrees,
		(double)(100.0 * S.m_TotalDegrees/1000.0));

	//Method 2: With Pointers
	FastUpdateTotalDegree(&S);
	printf("Total degrees of %s is %2.2lf (%2.2lf%%)\r\n",
		S.m_Name, S.m_TotalDegrees,
		(double)(100.0 * S.m_TotalDegrees/1000.0));
}
```


Passing Arrays and Pointers to Functions

Normal Array Passing

```c
#include "stdio.h"

void Sort(int values[], int nValues)
{
	int i, j, temp;
	for(i=0;i<nValues-1;i++)
		for(j=i;j<nValues;j++)
			if(values[i]>values[j])
			{
				temp = values[i];
				values[i] = values[j];
				values[j] = temp;
			}
}
void main()
{
	int i, values[5] = {89,73,28,94,32};
	Sort(values, 5);
	for(i=0;i<5;i++)
		printf("%d\r\n", values[i]);
}
```

Passing Arrays and Pointers to Functions

Array Passing with Pointers

```c
#include "stdio.h"

void Sort(int* values, int nValues)
{
	int i, j, temp;
	for(i=0;i<nValues-1;i++)
		for(j=i;j<nValues;j++)
			if(values[i]>values[j])
			{
				temp = values[i];
				values[i] = values[j];
				values[j] = temp;
			}
}

void main()
{
	int i, values[5] = {89,73,28,94,32};
	Sort(values, 5);
	for(i=0;i<5;i++)
		printf("%d\r\n", values[i]);
}
```

Method 2 is completely equivalent to Method 1 which means that:

`void Sort(int values[], int nValues)`	is equivelant to	`void Sort(int* values, int nValues)`

Programmer is free to choose which notation is sutable, because both methods gives the same behaviour.

Finally we can summarize function parameters types

- **1. Input Parameters (Calling by Value)**
  The parameter values is completely transmitted to the function. This gives the
  function the ability to read the transmitted data only.
- **2. Input/Output Parameters (Refrence or Pointer)**
  The parameter pointer (refernce) is transmitted only. This gives the function the
  ability to read from and write to the original parameters.
- **3. Output Parameters (Return Value)**

  The return data of the function is assumed as an output parameter. Normally C does

  not provide other Output parameters except the return value.

The sort function contains the two types of parameters.

```
void Sort(int* values /*input/output*/, int nValues /*input*/)

{
        int i, j, temp;
        for(i=0;i<nValues-1;i++)
                for(j=i;j<nValues;j++)
                        if(values[i]>values[j])
                        {
                                temp = values[i];
                                values[i] = values[j];
                                values[j] = temp;
                        }
}
```

values parameter is an input/output parameter.
nValues parameter is an input parameter.

Pointer with Unknown Type (void*)

- Programmer can define general pointer without specifying a linked data type.
- This type of pointers called (void pointer).
- void pointer can not be used normally to manipulated data, **it is required to type cast each data access operation.**

```c
#include "stdio.h"

void main()
{
        int x = 5;
        double y = 10.3;
        void* p;

        p = &x;
        *(int*)p = 8;
        printf("x = %d\r\n", x);

        p = &y;
        *(double*)p = 3.3;
        printf("y = %lf\r\n", y);
}
```

```
C:\Windows\system32\cmd.exe
x = 8
y = 3.300000
Press any key to continue . . .
```

Example: Universal Compare with void Pointers

```c
//Prepared by Eng.Keroles
#include <stdio.h>
int Compare(void* value1, void* value2, int type)
{
    int r;
    switch(type)
    {
    case 1://Integer
        if( *(int*)value1 == *(int*)value2 )r = 0;
        else if( *(int*)value1 > *(int*)value2 )r = 1;
        else r = -1;
        break;
    case 2://double
        if( *(double*)value1 == *(double*)value2 )r = 0;
        else if( *(double*)value1 > *(double*)value2 )r = 1;
        else r = -1;
        break;
    }
    return r;
}
```

Example: Universal Compare with void Pointers


```
22 int main(int argt ,char**argv) {
23     int x1 = 5, x2 = 6, x3 = 5;
24     double y1 = 10.3, y2 = 8.3, y3 = 11.9;
25     printf("Compare x1 and x2 gives %d\r\n",
26             Compare(&x1, &x2, 1));
27     printf("Compare x1 and x3 gives %d\r\n",
28             Compare(&x1, &x3, 1));
29     printf("Compare y1 and y2 gives %d\r\n",
30             Compare(&y1, &y2, 2));
31     printf("Compare y1 and y3 gives %d\r\n",
32             Compare(&y1, &y3, 2));
33     return 0   ;
34 }
```

Problems | AVR Supported MCUs | Tasks | Console

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\cou

```
Compare x1 and x2 gives -1

Compare x1 and x3 gives 0

Compare y1 and y2 gives 1

Compare y1 and y3 gives -1
```

The function prototype is:

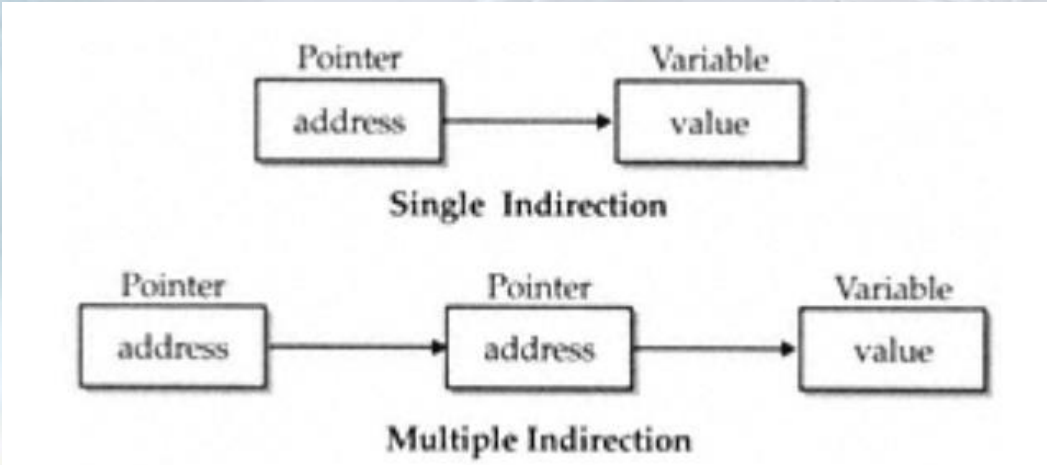
- **int Compare(void* value1, void* value2, int type)** which means it takes any two pointers with uncknown type, the third parameter informs the type of the submitted values (1 means integer, 2 means double).

Pointer Conversions
void * pointers

- In C, it is permissible to assign a **void *** pointer to any other type of pointer. It is also permissible to assign any other type of pointer to a **void *** pointer. A **void * pointer is called a *generic pointer***.
- The **void *** pointer is used to specify a pointer whose base type is unknown.
- The **void *** type allows a function to specify a parameter that is capable of **receiving any type of pointer argument without reporting a type mismatch**.
- It is also used to refer to raw memory (such as that returned by the **malloc( )** function described later in this chapter)

Multiple Indirection
Pointer to Pointer

- You can have a pointer point to another pointer that points to the target value.
- **This situation is called**
  multiple indirection, or pointers to pointers

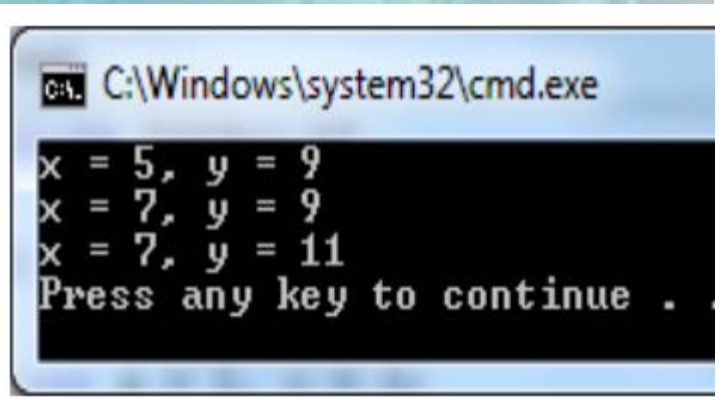


Using Pointer to Pointer


```c
#include "stdio.h"

void main()
{
	int x = 5, y = 9;
	int* pX = &x; //Pointer
	int** ppX = &pX; //Pointer to Pointer

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

	**ppX = 7;
	printf("x = %d, y = %d\r\n", x, y);

	*ppX = &y;

	*pX = 11;
	printf("x = %d, y = %d\r\n", x, y);
}
```

Using Pointer to Pointer


```c
#include "stdio.h"

void main()
{
	int x = 5, y = 9;
	int* pX = &x; //Pointer
	int** ppX = &pX; //Pointer to Pointer

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

	**ppX = 7;
	printf("x = %d, y = %d\r\n", x, y);

	*ppX = &y;

	*pX = 11;
	printf("x = %d, y = %d\r\n", x, y);
}
```



NULL and Unassigned Pointers

- If the pointer is unassigned it will contain an invalid address, it is unsafe to use an unassigned pointer, normally the program will crash. Fllowing program will crash because
the (pX) pointer is not pointed to a valid address, it contain a memory gurbage

```
#include "stdio.h"

void main()
{
        int* pX;
        printf("pX point to %d", *pX);
}
```

NULL and Unassigned Pointers

- To avoid using unassigned pointers, all pointers must hold a valid address, if not it must hold a zero value. Sometimes zero value called (NULL). Above program may be fixed as shown bellow:

```
#include "stdio.h"

void main()
{
        int* pX = NULL;
        if(pX!=NULL)
                printf("pX point to %d", *pX);
        else
                printf("pX is not initialized");
}
```

Labs

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10

An Illustration


```
int i = 5, j = 10;
int *ptr;    /* declare a pointer-to-integer variable */
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;  /* declare a pointer-to-pointer-to-integer variable */
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	
pptr	int **	integer pointer pointer variable	

Double

An Illustration

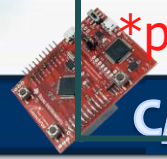

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;     /* store address-of i to ptr */
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	
*ptr	int	de-reference of ptr	5

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr; /* store address-of ptr to pptr */
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)



An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	3
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	3

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	7

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	10

#LEARN_IN DEPTH

#Be_professional_in embedded_system

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	9

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration


```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	-2
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	-2

Pointer Arithmetic

- What's `ptr + 1`?
  - → The next memory location!
- What's `ptr - 1`?
  - → The previous memory location!
- What's `ptr * 2` and `ptr / 2`?
  - → Invalid operations!!!

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	9.0

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	6.0

Pointer Arithmetic and Array


```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	7.0
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	7.0

Pointer to Function

- powerful feature of **C is the *function pointer***.
- A function has a **physical location in memory** that can be assigned to a pointer.
- This address is the entry point of the function and it is the address used when the function is called.
- Once a pointer points to a function, the function can be called through that pointer.
- Function pointers also allow functions to be passed as arguments to other functions

Example 1

pointers.c

```c
/*
 * pointers.c
 *
 *  Created on: Oct 15, 2020
 *      Author: Keroles Shenouda
 */

#include "stdio.h"

void (* PFun)() ;

void print_online_diploma ()
{
    printf ("LEARN IN Depth \n");
}
int main ()
{
    print_online_diploma ();
    PFun = print_online_diploma ;
    PFun();
    return 0 ;
}
```

Variables | Breakpoints | Modules | Expressions | Interactive Console

Expression	Type	Value
PFun	void (*)()	0x4013f0 <print_online_diploma>
Add new expression		

Disassembly

Enter location here

```
print_online_diploma:
004013f0:   push   %ebp
004013f1:   mov    %esp,%ebp
004013f3:   sub    $0x8,%esp
14            printf ("LEARN IN Depth \n");
004013f6:   movl   $0x403030,(%esp)
004013fd:   call   0x4015f0 <puts>
15          }
00401402:   leave
00401403:   ret
17          {
main:
00401404:   lea    0x4(%esp),%ecx
00401408:   and    $0xfffffff0,%esp
0040140b:   pushl  -0x4(%ecx)
0040140e:   push   %ebp
```

Browser Output

main.s | *main.c

Example 2

```c
//Prepared by Eng.Keroles
#include <stdio.h>
#include <string.h>
//prototype
void check(char *a, char *b, int (*cmp)(const char *, const char *));

int main(int argt ,char**argv) {
    char s1[80], s2[80];
    int (*p)(const char *, const char *); /* function pointer */
    p = strcmp; /* assign address of strcmp to p */
    printf("Enter two strings.\n");
     fflush(stdin); fflush(stdout);
    gets(s1);
     fflush(stdin); fflush(stdout);
    gets(s2);
    check(s1, s2, p); /* pass address of strcmp via p */
    return 0   ;
}
void check(char *a, char *b, int (*cmp) (const char *, const char *))
{
    printf("Testing for equality.\n");
            if(!(*cmp)(a, b)) printf("Equal");
            else printf("Not Equal");
}
```

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Problems AVR Supported MCUs Tasks Console

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\cours

```
Enter two strings.
a
b
Testing for equality.
Not Equal
```

Problems AVR Supported MCUs Tasks Console Proper

<terminated> (exit value: 0) session2.exe [C/C++ Application] D:\courses\C_Cours

```
Enter two strings.
a
a
Testing for equality.
Equal
```

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

main.s | main.c

```
5  void check(char *a, char *b, int (*cmp)(const char *, const char *));
6
7  int main(int argt ,char**argv) {
8      char s1[80], s2[80];
9      int (*p)(const char *, const char *); /* function pointer */
10     p = strcmp; /* assign address of strcmp to p */
11     printf("Enter two strings.\n");
12      fflush(stdin); fflush(stdout);
13     gets(s1);
14      fflush(stdin); fflush(stdout);
```

Console | Tasks | Problems | Executables | Memory

ssion2.exe [C/C++ Application] session2.exe

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

Name	Type	Value
(x)= argt	int	1
argv	char **	0x6546f8
s1	char [80]	0x61fedc
s2	char [80]	0x61fe8c
p	int (*)(const char *, const char *)	0x403a78 <strcmp>

Pointers Tricks

- How to Read C complex pointer
- double to integer
- Modularity
- Pointer with Constant

How to Read C complex pointer expression

Before We Learn How to Read Complex Array we should first know precedence and associative .

•**Priority** : Operator precedence describes the order in which C reads expressions

•**order** : Order operators of equal precedence in an expression are applied

Before Reading Article You Should know Precedence and Associative Table

Operator	Precedence	Associative
(),[]	1	Left to Right
*,Identifier	2	Right to Left
Data Type	3	–

How to Read C complex pointer expression

Different Terms From Table –

()	Bracket operator **OR** function operator.
[]	Array subscription operator
*****	Pointer operator
Identifier	**Name of Pointer Variable**
Data type	Type of Pointer

How to Read C complex pointer expression

- char (* ptr)[5]

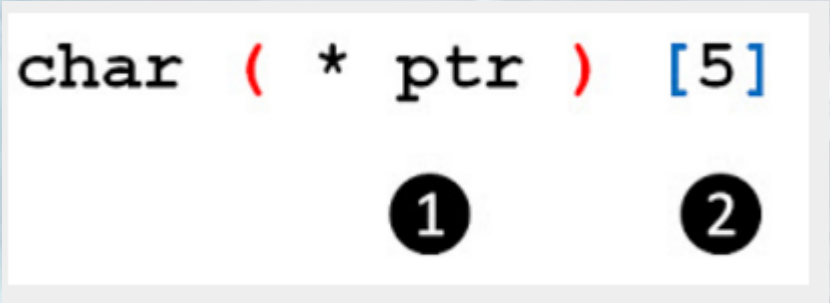
Step 1 :

•Observe Above Precedence Table

•**[] and () have Same Priority**

•Using Associativity , **Highest Priority** Element is decided

•Associativity is from Left to Right **First Priority is Given to "()"**



How to Read C complex pointer expression

Step 2 :

- Between Pair of Brackets again we have to decide which one has highest priority ? '*' or 'ptr' ?
- *** and Identifier ptr have Same Priority**
- Associativity is from Right to Left **First Priority is Given to "ptr"**

```
char ( * ptr ) [5]
       2   1
```

How to Read C complex pointer expression

Read it as –

ptr is pointer to a one dimensional array having size five which can store data of type char

Step 3 :

- ✔ **Assign third Priority to [].**
- ✔ Write 3 under []

Step 4 :

- ✔ Here Char is Data Type
- ✔ Assign **Priority 4 to char**

C is easy

C isn't that hard:

`void (*(*f[])())() defines f as an array of unspecified size, of`

f as array of pointer to function returning pointer to function returning void

#LEARN_IN DEPTH

#Be_professional_in embedded_system

C is easy

```
void ( * ( *f [] ) () ) ()
```


How to Read C complex pointer expression examples

```
int (*p)(char);
```

```

```

```
char ** (*p)(float, float);
```

```

```

```
void * (*a[5])(char * const, char * const);
```

How to Read C complex pointer expression examples

```
int (*p)(char);
```

This declares `p` as a pointer to a function that takes a `char` argument and returns an `int`.

A pointer to a function that takes two `float`s and returns a pointer to a pointer to a `char` would be declared as:

Hide Copy Code

```
char ** (*p)(float, float);
```

How about an array of 5 pointers to functions that receive two `const` pointers to `char`s and return a `void` pointer?

Hide Copy Code

```
void * (*a[5])(char * const, char * const);
```

How to Read C complex pointer expression examples

```
int * (* (*fp1) (int) ) [10];
```

This can be interpreted as follows:

1. Start from the variable name ---------- `fp1`
2. Nothing to right but `)` so go left to find `*` ---------- is a pointer
3. Jump out of parentheses and encounter (`int`) --------- to a function that takes an `int` as argument
4. Go left, find `*` ---------- and returns a pointer
5. Jump put of parentheses, go right and hit `[10]` -------- to an array of 10
6. Go left find `*` ---------- pointers to
7. Go left again, find `int` ---------- `int`s.

How to Read C complex pointer expression examples

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

1. Start from the variable name ---------- arr
2. Go right, find array subscript ---------- is an array of 5
3. Go left, find * ---------- pointers
4. Jump out of parentheses, go right to find () ------ to functions
5. Go left, encounter * ---------- that return pointers
6. Jump out, go right, find () ---------- to functions
7. Go left, find * ---------- that return pointers
8. Continue left, find int ---------- to ints.

How to Read C complex pointer expression examples

- Test Your self

```
int i;                  an int
int *p;                 an int pointer (ptr to an int)
int a[];                an array of ints
int f();                a function returning an int
int **pp;               a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];            a pointer to an array of ints
int (*pf)();            a pointer to a function returning an int
int *ap[];              an array of int pointers (array of ptrs to ints)
int aa[][];             an array of arrays of ints
int af[]();             an array of functions returning an int (ILLEGAL)
int *fp();              a function returning an int pointer
int fa()[];             a function returning an array of ints (ILLEGAL)
int ff()();             a function returning a function returning an int
                                (ILLEGAL)
int ***ppp;             a pointer to a pointer to an int pointer
int (**ppa)[];          a pointer to a pointer to an array of ints
int (**ppf)();          a pointer to a pointer to a function returning an int
int *(*pap)[];          a pointer to an array of int pointers
int (*paa)[][];         a pointer to an array of arrays of ints
int (*paf)[]();         a pointer to a an array of functions returning an int
                                (ILLEGAL)
int *(*pfp)();          a pointer to a function returning an int pointer
int (*pfa)()[];         a pointer to a function returning an array of ints
                                (ILLEGAL)
int (*pff)()();         a pointer to a function returning a function
                                returning an int (ILLEGAL)
int **app[];            an array of pointers to int pointers
int (*apa[])[];         an array of pointers to arrays of ints
int (*apf[])();         an array of pointers to functions returning an int
int *aap[][];           an array of arrays of int pointers
int aaa[][][];          an array of arrays of arrays of ints
int aaf[][]();          an array of arrays of functions returning an int
                                (ILLEGAL)
int *afp[]();           an array of functions returning int pointers (ILLEGAL)
int afa[]()[];          an array of functions returning an array of ints
                                (ILLEGAL)
int aff[]()();          an array of functions returning functions
                                returning an int (ILLEGAL)
int **fpp();            a function returning a pointer to an int pointer
int (*fpa())[];         a function returning a pointer to an array of ints
int (*fpf())();         a function returning a pointer to a function
                                returning an int
int *fap()[];           a function returning an array of int pointers (ILLEGAL)
int faa()[][];          a function returning an array of arrays of ints
                                (ILLEGAL)
int faf()[]();          a function returning an array of functions
                                returning an int (ILLEGAL)
int *ffp()();           a function returning a function
                                returning an int pointer (ILLEGAL)
```

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Pointers Tricks

- How to Read C complex pointer
- Modularity
- Pointers on Embedded C
- Pointer with Constant

Pointers on Embedded C

How to read 4 Bytes by Bytes From stream of data ?

What is the Output ?

main.c | _register_frame_info() at 0x401285

```c
/*
 * main.c
 *
 *  Created on: Jun 16, 2017
 *      Author: Keroles Shenouda
 */


#include "stdio.h"

int main ()
{
    unsigned char x[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    unsigned int* y ;
    unsigned short* z = (unsigned short*)x ;
    unsigned long long* d = (unsigned long long*) x ;
    y = (unsigned int*) x;
    printf ("y=0x%x \n",*y);
    y++ ;
    printf ("y=0x%x \n",*y);
    y++ ;
    printf ("y=0x%x \n",*y);
    y++ ;
    printf ("y=0x%x \n",*y);
    ////////////////////////////////
    printf ("z=0x%x \n",*z);
    z++ ;
    printf ("z=0x%x \n",*z);
    z++ ;
    printf ("z=0x%x \n",*z);
    z++ ;
    printf ("z=0x%x \n",*z);
    ////////////////////////////////
    printf ("d=0x%llx \n",*d);
    d++ ;
    printf ("d=0x%llx \n",*d);
    return 0 ;
}
```

FOLLOW US
Press here
88

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

ook.com/groups/embedded.system.KS/

What is the Output ?

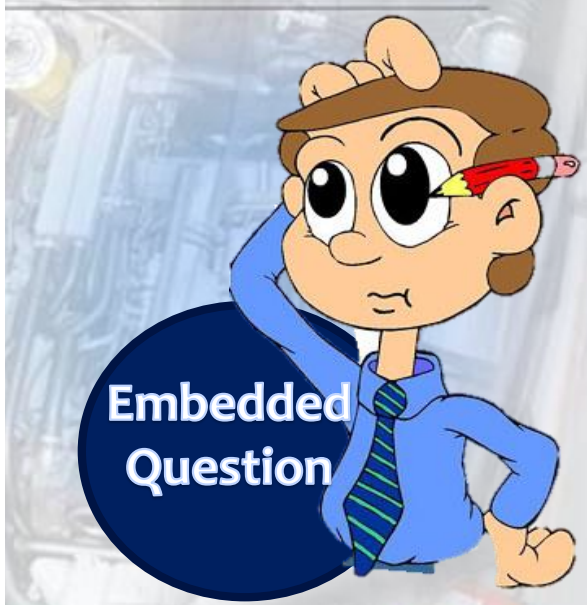
main.c | _register_frame_info() at 0x401285

```c
/*
 * main.c
 *
 *  Created on: Jun 16, 2017
 *      Author: Keroles Shenouda
 */


#include "stdio.h"

int main ()
{
    unsigned char x[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    unsigned int* y ;
    unsigned short* z = (unsigned short*)x ;
    unsigned long long* d = (unsigned long long*) x ;
    y = (unsigned int*) x;
    printf ("y=0x%x \n",*y);
    y++ ;
    printf ("y=0x%x \n",*y);
    y++ ;
    printf ("y=0x%x \n",*y);
    y++ ;
    printf ("y=0x%x \n",*y);
    ////////////////////////////////
    printf ("z=0x%x \n",*z);
    z++ ;
    printf ("z=0x%x \n",*z);
    z++ ;
    printf ("z=0x%x \n",*z);
    z++ ;
    printf ("z=0x%x \n",*z);
    ////////////////////////////////
    printf ("d=0x%llx \n",*d);
    d++ ;
    printf ("d=0x%llx \n",*d);
    return 0 ;
}
```

Problems | AVR Supported MCUs

<terminated> (exit value: 0) session_3.exe [C

```
y=0x4030201
y=0x8070605
y=0xc0b0a09
y=0x100f0e0d
z=0x201
z=0x403
z=0x605
z=0x807
d=0x807060504030201
d=0x100f0e0d0c0b0a09
```



FOLLOW US

Press here

89

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

ook.com/groups/embedded.system.KS/

Pointers Tricks

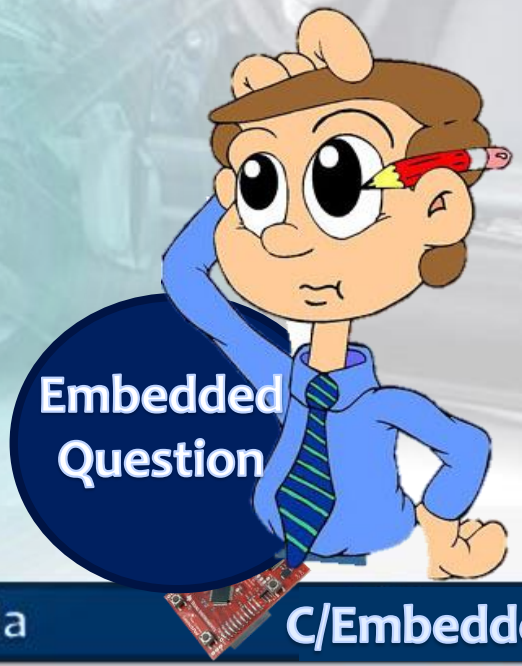
- How to Read C complex pointer
- Modularity
- Pointers on Embedded C
- Pointer with Constant

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Modularity

What is the benefit of the pointer to function on embedded C (Modularity)?



APP

↓

Hardware Abstraction Layer

↓

Drivers

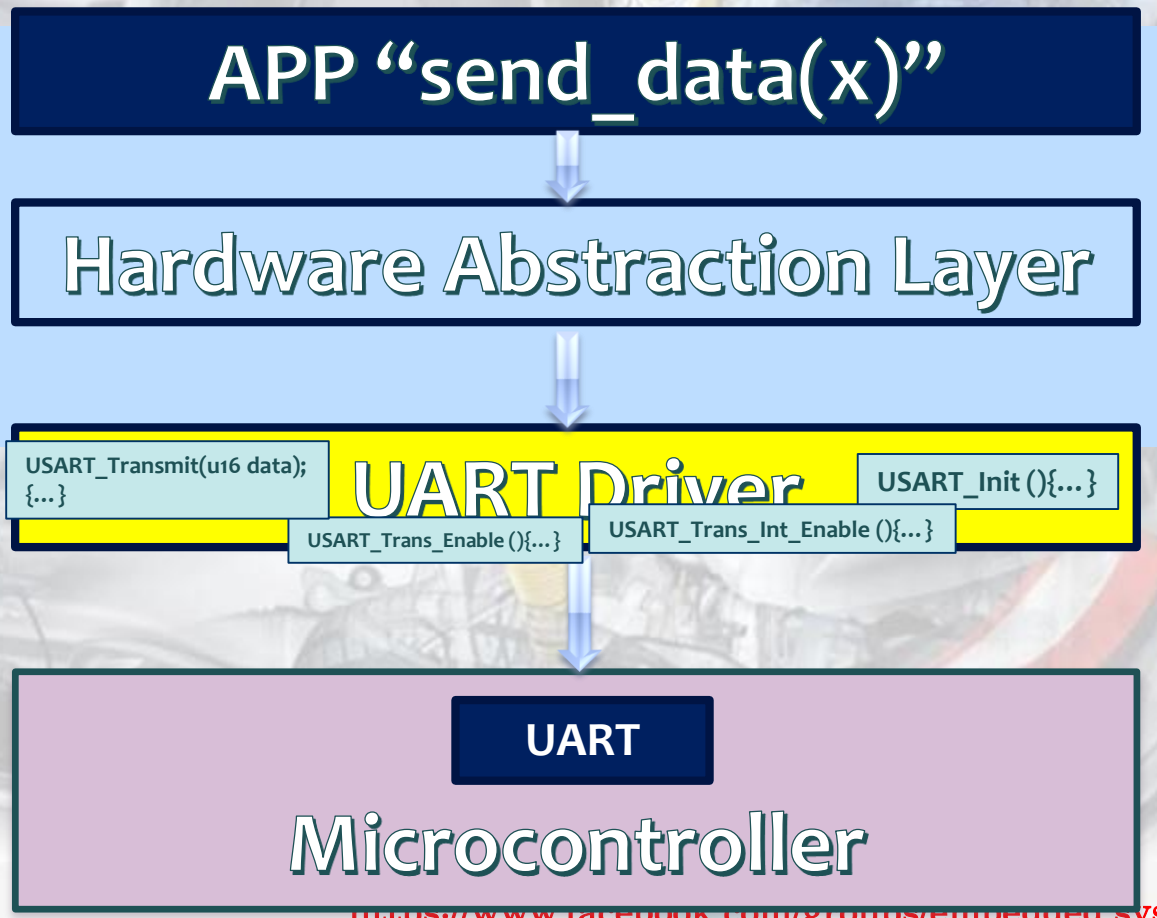
↓

Microcontroller

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

For example “UART” (send data)

```
Send_data(u16 data)
{
USART_Init ();
USART_Trans_Int_Enable();
 USART_Trans_Enable();
USART_Transmit(u16 data);
}
```



#LEARN_IN DEPTH

#Be_professional_in_embedded_system

For example “UART” (send data)

```
Send_data(u16 data)
{
USART_Init ();
USART_Trans_Int_Enable();
 USART_Trans_Enable();
USART_Transmit(u16 data);
}
```

APP “send_data(x)”

Hardware Abstraction Layer

USART_Transmit(u16 data); {…}

UART Driver

USART_Init (){…}

USART_Trans_Enable (){…}

USART_Trans_Int_Enable (){…}

Byte → Send data → Byte

UART

Microcontroller

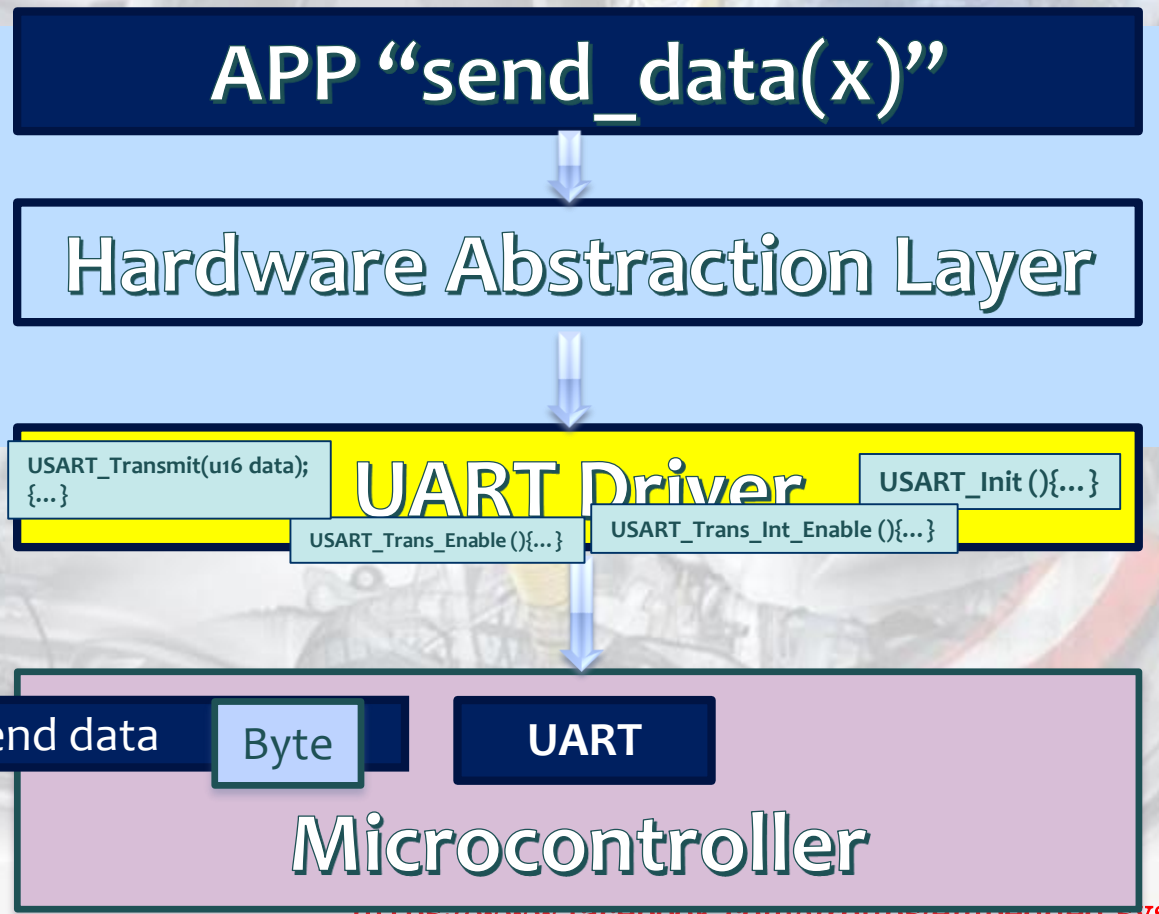
Once the UART sent the Data, The TX interrupt will insert and the CPU will call

```
ISR(USART_TXC_vect)
{ ……. }
```

For example “UART” (send data)

How to sense the interrupt from APP without depend on the Mic. ?

```
Send_data(u16 data)
{
USART_Init ();
USART_Trans_Int_Enable();
 USART_Trans_Enable();
USART_Transmit(u16 data);
}
```



Once the UART sent the Data, The TX interrupt will insert and the CPU will call

```
ISR(USART_TXC_vect)
{  ……..  }
```

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

By the pointer to function


```
void (*Ptr_To_Trans_Int) (void);

extern void USART_callback_Trans_Int(void (*Ptr_to_Func)(void))
{
Ptr_To_Trans_Int = Ptr_to_Func;
ISR(USART_TXC_vect)
{
        (*Ptr_To_Trans_Int)();
}
```

APP “send_data(x)”

Hardware Abstraction Layer

UART Driver

Byte | Send data | Byte | UART

Microcontroller

ISR(USART_TXC_vect)
{ ..... }

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

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Pointers Tricks

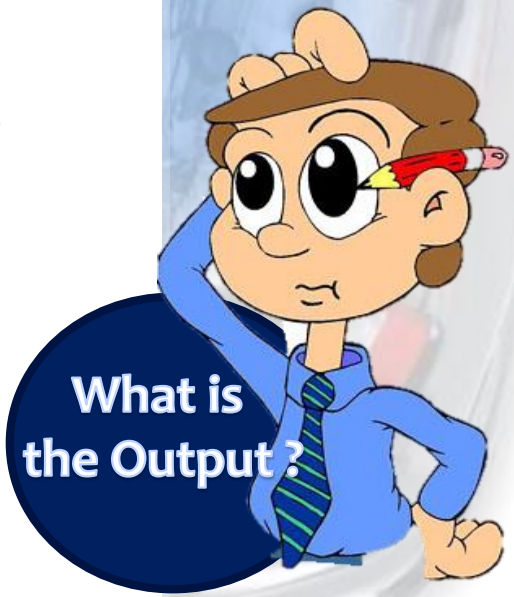
- How to Read C complex pointer
- Modularity
- Pointers on Embedded C
- Pointer with Constant

Variable Quantifiers - const

- const is used with a datatype declaration or definition to specify an unchanging value
- const objects may not be changed

main.c

```c
/*
 * main.c
 *
 *  Created on: Jun 16, 2017
 *      Author: Keroles Shenouda
 */


#include "stdio.h"

int main ()
{
    const int five = 5;
 const double pi = 3.141593;
    pi = pi + 1;
    return 0 ;
}
```



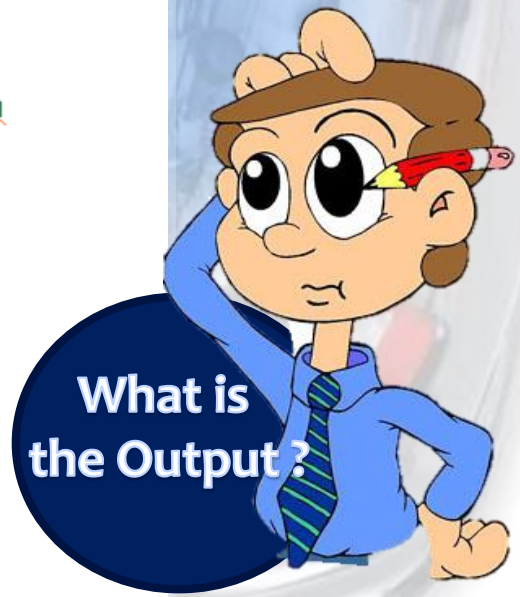
Variable Quantifiers - const

- const is used with a datatype declaration or definition to specify an unchanging value
- const objects may not be changed

```
Problems  AVR Supported MCUs  Tasks  Console  Properties  AVR Device Explorer
CDT Build Console [session_3]
14:45:49 **** Incremental Build of configuration Debug for project session_3
Info: Internal Builder is used for build
gcc -O0 -g3 -Wall -c -fmessage-length=0 -o main.o "..\\main.c"
..\main.c: In function 'main':
..\main.c:15:5: error: assignment of read-only variable 'pi'
  pi = pi + 1;
     ^
..\main.c:13:12: warning: unused variable 'five' [-Wunused-variable]
  const int five = 5;
            ^

14:45:49 Build Finished (took 311ms)
```

main.c

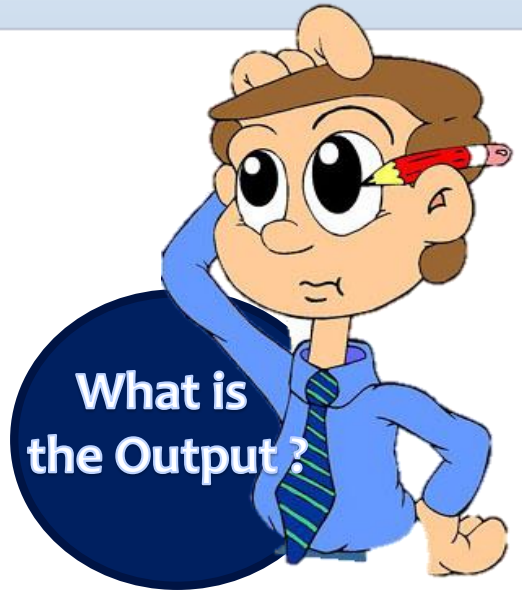
```c
/*
 * main.c
 *
 *  Created on: Jun 16, 2017
 *      Author: Keroles Shenouda
 */


#include "stdio.h"

int main ()
{
    const int five = 5;
 const double pi = 3.141593;
    pi = pi + 1;
    return 0 ;
}
```



Constant with pointer

main.c

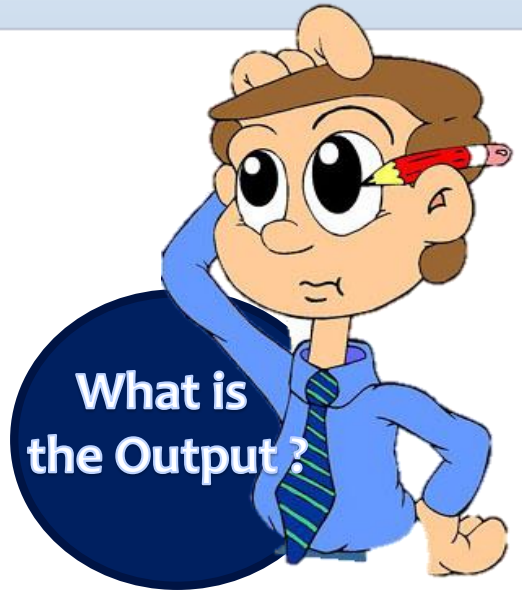
```
 *  Created on: Jun 16, 2017
 *      Author: Keroles Shenouda
 */


#include "stdio.h"

int main ()
{
    //Definition of the variable
    int a = 10;

    //Definition of pointer to constant
    const int* ptr = &a; //Now, ptr is pointing to the value of the variable a

    *ptr = 30;
    printf ("%d",*ptr);

    return 0 ;
}
```



Constant with pointer

main.c

```
 *  Created on: Jun 16, 2017
 *      Author: Keroles Shenouda
 */


#include "stdio.h"

int main ()
{
    //Definition of the variable
    int a = 10;

    //Definition of pointer to constant
    const int* ptr = &a; //Now, ptr is pointing to the value of the variable a

    *ptr = 30;
    printf ("%d",*ptr);

    return 0 ;
}
```



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

CDT Build Console [session_3]

```
14:49:50 **** Incremental Build of configuration Debug for project session_3 ****
Info: Internal Builder is used for build
gcc -O0 -g3 -Wall -c -fmessage-length=0 -o main.o "..\\main.c"
..\main.c: In function 'main':
..\main.c:19:7: error: assignment of read-only location '*ptr'
  *ptr = 30;
       ^

14:49:50 Build Finished (took 306ms)
```

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Complete the table yes/no …

Example	Value constant	Pointer Constant
char *ptr		
const char *ptr		
char const *ptr		
char* const ptr		
const char *const ptr		

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

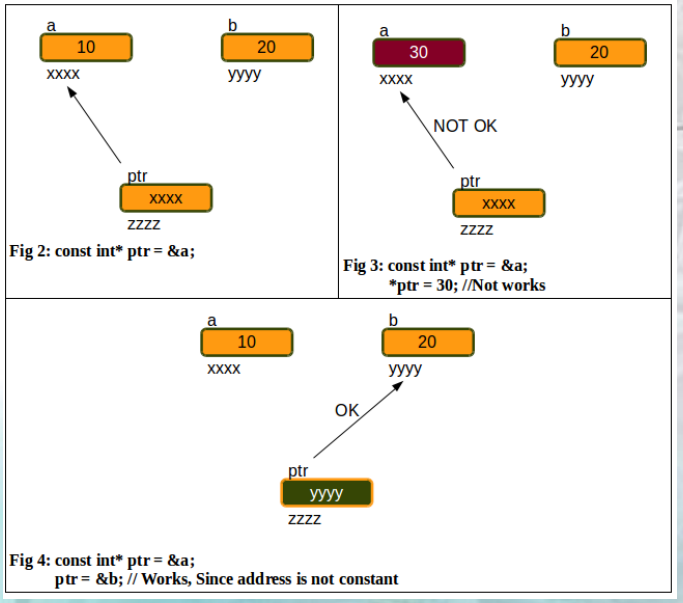


Fig 2: const int* ptr = &a;

Fig 3: const int* ptr = &a;
*ptr = 30; //Not works

Fig 4: const int* ptr = &a;
ptr = &b; // Works, Since address is not constant

```
int* ptr = &a;
```

constant

```
int const *ptr = &a;
const int *ptr = &a;
```

Value is constant

```
int* const ptr = &a;
```

Pointer is constant

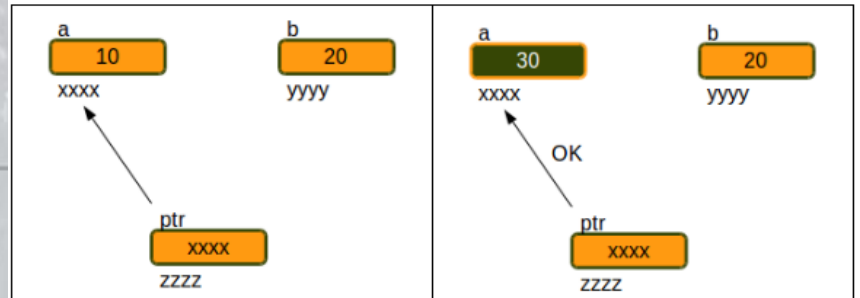


Fig 6: int* const ptr = &a;

Fig 7: int* const ptr = &a;
*ptr = 30; // works

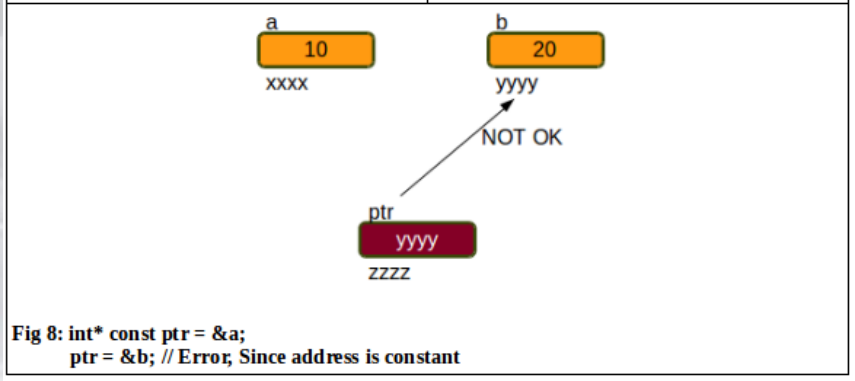


Fig 8: int* const ptr = &a;
ptr = &b; // Error, Since address is constant

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Complete the table yes/no ...

Example	Value constant	Pointer Constant
char *ptr	Try	Now
const char *ptr		
char const *ptr		
char* const ptr		
const char *const ptr		

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

Complete the table yes/no ...

Example	Value constant	Pointer Constant
char *ptr	No	*No*
const char *ptr	Yes	*No*
char const *ptr	Yes	*No*
char* const ptr	No	*Yes*
const char *const ptr	*Yes*	*Yes*

Interview Tricks: Pointer vs Array in C

- Most of the time, **pointer** and **array** accesses **can be treated** as acting the same, the major exceptions being:
- the sizeof operator
  - sizeof(array) returns the amount of memory used by all elements in array
  - sizeof(pointer) only returns the amount of memory used by the pointer variable itself
- the & operator
  - &array is an alias for &array[0] and returns the address of the first element in array
  - &pointer returns the address of pointer
- a string literal initialization of a character array
  - char array[] = "abc" sets the first four elements in array to 'a', 'b', 'c', and '\0'
  - char *pointer = "abc" sets pointer to the address of the "abc" string (which may be stored in read-only memory and thus unchangeable)
- Pointer variable can be assigned a value whereas array variable cannot be.
- Arithmetic on pointer variable is allowed.
  - p++; /*Legal*/
  - a++; /*illegal*/

```
int a[10];
int *p;
p=a; /*legal*/
a=p; /*illegal*
```

#LEARN_IN DEPTH

#Be_professional_in_embedded_system

What will the output and why ?

pointers.c

```
 4  *  Created on: Oct 15, 2020
 5  *      Author: Keroles Shenouda
 6  */
 7
 8 #include "stdio.h"
 9
10 void (* PFun)() ;
11
12 void print_online_diploma ()
13 {
14     printf ("LEARN IN Depth \n");
15 }
16 int main ()
17 {
18     char *pmessage = "now is the time";
19     printf ("%s \n",pmessage);
20
21     pmessage++ ;
22     printf ("%s \n",pmessage);
23
24     int x = sizeof (pmessage);
25     printf ("%d \n",x);
26
27
28     *pmessage = 'S';
29     printf ("%s \n",pmessage);
30
31
32     return 0 ;
33 }
34
```

What happens if we use out of bounds index in an array in C language?

pointers.c

```c
#include <stdio.h>

int main(void)
{
    int arr[5];
    int  i;

    arr[0] = 10;  //valid
    arr[1] = 20;  //valid
    arr[2] = 30;  //valid
    arr[3] = 40;  //valid
    arr[4] = 50;  //valid
    arr[5] = 60;  //invalid (out of bounds index)

    //printing all elements
    for( i=0; i<6; i++ )
        printf("arr[%d]: %d\n",i,arr[i]);

    return 0;
}
```

Output

```
arr[0]: 10 arr[1]: 20 arr[2]: 30 arr[3]: 40
arr[4]: 50 arr[5]: 11035
```

Explanation:

In the program, array size is 5, so array indexing will be from `arr[0]` to `arr[4]`. But, Here I assigned value 60 to `arr[5]` (`arr[5]` **index is out of bounds array index**). Program compiled and executed successfully, but while printing the value, value of `arr[5]` is unpredictable/garbage. I assigned 60 in it and the result is 11035 (which can be anything).

#LEARN_IN DEPTH

#Be_professional_in embedded_system

Describe each pointer ☺

- Null Pointer
- Dangling Pointer
- Generic Pointer
- Wild Pointer
- Complex Pointer
- Near Pointer
- Far Pointer
- Huge Pointer

Null Pointer

- Pointer which is pointing to nothing
- Pointers which is initialized to NULL
- EX:
  - Float* ptr = (float*)0; //0 is a null
  - Float* ptr = NULL ;

pointers.c

```c
//Keroles Shenouda
//www.learn-in-depth.com
#include<stdio.h>

int main()
{
    int* ptr =NULL ;
    printf ("%u \n",ptr);
    return 0;
}

```


Null Pointer

- Pointer which is pointing to nothing
- Pointers which is initialized to NULL
- EX:
  - Float* ptr = (float*)0; //0 is a null
  - Float* ptr = NULL ;

pointers.c

```
1  //Keroles Shenouda
2  //www.learn-in-depth.com
3  #include<stdio.h>
4
5  int main()
6  {
7      int* ptr =NULL ;
8      printf ("%u \n",ptr);
9      return 0;
10 }
11
```

Problems Tasks Console Properties

<terminated> (exit value: 0) pointers_lab.exe [C/C++ Application] D:\courses\new_

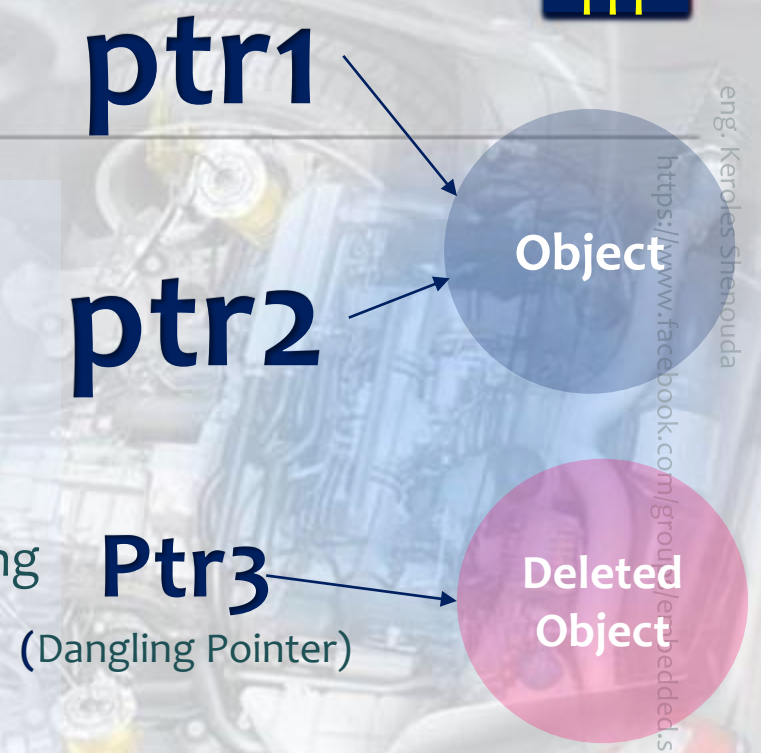
```
0
```

Dangling Pointer

- Dandling Pointer arise when an object is deleted or de-allocated

Without modifying the value of the pointer.

- Pointer which points to the memory location of the de-allocated memory.
- Pointer pointing to non-existing memory location is called dangling pointer.



Dangling Pointer Example

pointers.c

```c
//Keroles Shenouda
//www.learn-in-depth.com
#include<stdio.h>

int main()
{
    int* ptr = malloc(4) ;
//  ....
//  ....
//  ....
    free (ptr); //ptr now becomes a dangling pointer
    return 0;
}

```


Dangling Pointer Example

The y is invisible in outer block,
Then the pointer is still pointing in the same invalid memory Location in outer block then the pointer becomes dangling

pointers.c

```c
//Keroles Shenouda
//www.learn-in-depth.com
#include<stdio.h>

int main()
{
    char* ptr = NULL ;

    {

        char y =0 ;
        ptr = &y ;
    }

    //ptr now is dangling pointer


    return 0;
}

```

Generic Pointer

- The pointer which is define by
  - void* Ptr

- We are talking on it in details in the previous slides

Wild pointer

- A pointer in C that has not been initialized till its first use is knows as the wild pointer.
- Points to some random memory location.
- To avoid it, initialize it with the declaration

pointers.c

```c
//Keroles Shenouda
//www.learn-in-depth.com
#include<stdio.h>

int main()
{
    char* ptr  ;

    //ptr is a wild pointer
    printf ("%d \n",*ptr);

    return 0;
}

```

Complex pointer

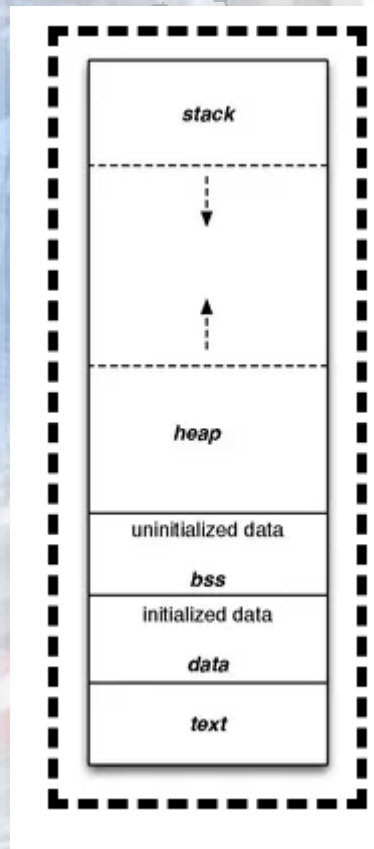
```
int i;              an int
int *p;             an int pointer (ptr to an int)
int a[];            an array of ints
int f();            a function returning an int
int **pp;           a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];        a pointer to an array of ints
int (*pf)();        a pointer to a function returning an int
int *ap[];          an array of int pointers (array of ptrs to ints)
int aa[][];         an array of arrays of ints
int af[]();         an array of functions returning an int (ILLEGAL)
int *fp();          a function returning an int pointer
int fa()[];         a function returning an array of i
int ff()();         a function returning a function re
                           (ILLEGAL)
int ***ppp;         a pointer to a pointer to an int p
int (**ppa)[];      a pointer to a pointer to an array
int (**ppf)();      a pointer to a pointer to a functi
int *(*pap)[];      a pointer to an array of int point
int (*paa)[][];     a pointer to an array of arrays of
int (*paf)[]();     a pointer to a an array of functic
                           (ILLEGAL)
int *(*pfp)();      a pointer to a function returning
int (*pfa)()[];     a pointer to a function returning an array of ints
                    (ILLEGAL)
```

Operator	Precedence	Associative
(),[]	1	Left to Right
*,Identifier	2	Right to Left
Data Type	3	–

ptr

Near, far and huge pointers

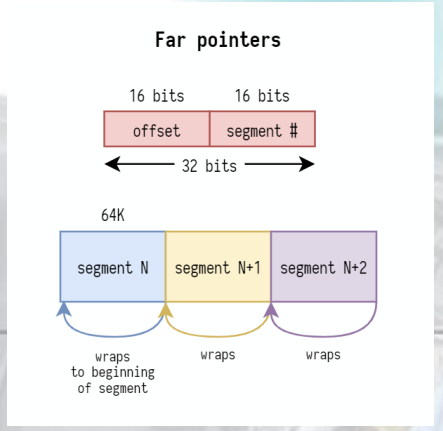
- This is not standard C syntax. This was an added feature for < 32-bit compilers.
- They are not needed and thus not supported on the modern compilers.
- The implementation of this pointers may vary from compiler to a compiler.
- Nowadays in the 32/64 bit world this keyword has become obsolete and useless.
- The idea about this pointers can be traced back computer CPUs were having a very small address and RAM.
- "near", "far", and "huge" were extensions to C to allow the programmer to give hints to the compiler about how to implement a pointer in the horrific segmented memory architecture of the Intel 8086/8088 (and 80186) CPUs. Those CPUs used two 16 bit registers



#LEARN_IN DEPTH

#Be_professional_in embedded_system

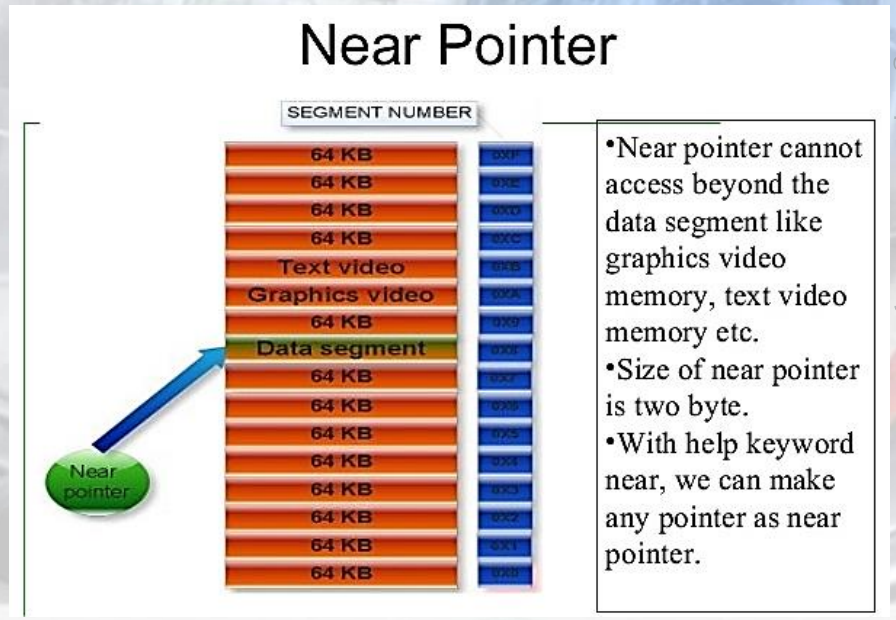
Near pointer



The pointer which can points only 64KB data segment

That is near pointer cannot access beyond the data segment like graphics video memory, text video memory, etc. Size of near pointer is two byte.

```
#include<stdio.h>

    int main()
{                                          Output: 2
    int x=25;
    int near* ptr;
    ptr=&x;
    printf(%d,sizeof ptr);
    return 0;
}
```



#LEARN_IN DEPTH

#Be_professional_in embedded_system

far Pointer

The pointer which can point or access whole the residence memory of RAM.

Size of the far pointer is 4 byte or 32 bit.

Example:

```
#include<stdio.h>
int main()
{
int x=10;
int far *ptr;
ptr=&x;
printf("%d",sizeof ptr);
return 0;
}
```

Output : 4

Address (Seg << 4)+Off(16bit) = effective address(20bit)

Segment	Offset	=	Effective
0x4000	0x0100	=	0x40100
0x4010	0x0000	=	0x40100
0x4FFF	0x0100	=	0x 500F0

Far Pointer

Huge pointer

The pointer which can point or access whole the residence memory of RAM

Size of the far pointer is 4 byte or 32 bit.

Like far pointer, **huge pointer** is also typically 32 bit and can access outside segment. In case of far pointers, a segment is fixed. In far pointer, the segment part cannot be modified, but in Huge it can be modified

#LEARN_IN DEPTH

#Be_professional_in embedded_system

study well

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

References

- C The Complete Reference 4th Ed Herbert Schildt
- A Tutorial on Data Representation
- std::printf, std::fprintf, std::sprintf, std::snprintf…..
- C Programming **for Engineers, Dr. Mohamed Sobh**
- **C programming expert.**
- fresh2refresh.com/c-programming
- **C programming Interview questions and answers**
- **C – Preprocessor directives**
- **Constant Pointers and Pointers to Constant A Subtle Difference in C Programming**