# Pointers

https://www.emertxe.com/embedded-systems/c-programming/c-sample-programs/

## Chapter 3 : Pointers Part 1

### P1: Pointer Basics - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra cómo definir y utilizar un puntero.

Source Code:

```
/*-----------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t078_ch3_pointers.c
 *   Title          : Pointer Basics - Example 1
 *   Description    : The C basic data types allows us to access a limited and fixed number of
 *                    the memory bytes. In order to increase the flexibility to access a range
 *                    of memory bytes C supports pointers. Any variable defined as a pointer
 *                    will hold an address of memory location. Have to be careful with dealing
 *                    the address ranges, else would lead to illegal memory access. This example
 *                    shows how to define and use a pointer
 *-----------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 5;
    /* ptr is a pointer which will be pointing to an integer */
    int *ptr;

    /*
    * As mentioned above the pointer should hold an address of an memory.
    * Now since this code is going to run in OS, we cannot store any random address
    * in it. So we have to define an variable and let the OS to allocate a memory for it.
    * An address of an variable is obtained by the & operator (called as referencing
    * operator)
    * Storing the address of x in ptr (So made it to point to x now)
    */
    ptr = &x;

    /* Print the address of x */
    printf("&x = %pn", &x);
    /*
    * Print the value of ptr. Yes you read it right. The address of x is stored
    * as value in ptr
    * And obviously ptr too, would have its own address which can be requested with &
    * as shown below in line 45
    */
    printf("ptr = %pn", ptr);

    /* Print the address of ptr */
    printf("&ptr = %pn", &ptr);
```

```
    return 0;
}
```

## P2: Pointers Basics - Example 2

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra cómo definir y utilizar un puntero y cómo acceder (desreferenciar) a un valor apuntado por un puntero.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t079_ch3_pointers.c
 *   Title          : Pointer Basics - Example 2
 *   Description    : This example shows how to define and use a pointer and how to access
 *                    (dereference) a value pointed by a pointer.
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 5;
    /* ptr is a pointer which will be pointing to an integer */
    int *ptr;

    /*
    * As mentioned above the pointer should hold an address of an memory.
    * Now since this code is going to run in OS, we cannot store any random address
    * in it. So we have to define an variable and let the OS to allocate a memory for it.
    * An address of an variable is obtained by the & operator (called as referencing operator)
    * Storing the address of x in ptr (So made it to point to x now)
    */
    ptr = &x;

    /* Print the value of x */
    printf("x = %dn", x);

    /*
    * Now to print the value at address held by the pointer, we need to use the
    * Dereferencing operator * ( Please don't get confused with * of comment ;) )
    */
    printf("Value at address pointed by ptr = %dn", *ptr);

    return 0;
}
```

## P3: Pointers Basics - Example 3

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros.

Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra cómo definir y utilizar un puntero y cómo acceder (desreferenciar) a un valor apuntado por un puntero.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t080_ch3_pointers.c
 *   Title          : Pointer Basics - Example 3
 *   Description    : This example shows how to define and use a pointer and how to access
 *                    (dereference) a value pointed by a pointer.
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 5;
    /* ptr is a pointer which will be pointing to an integer */
    int *ptr;

    /*
    * As mentioned above, the pointer should hold an address of an memory.
    * Now since this code is going to run in OS, we cannot store any random address
    * in it. So we have to define an variable and let the OS to allocate a memory for it.
    * An address of an variable is obtained by the & operator (called as referencing operator)
    * Storing the address of x in ptr (So made it to point to x now)
    */
    ptr = &x;

    /* Print the value of x using direct and indirect addressing */
    printf("x = %dn", x);
    printf("Value at address pointed by ptr = %dn", *ptr);

    /* Modify the value at address pointed by ptr */
   *ptr = 20;

    /* The value of x will be changed to 20 */
    printf("x = %dn", x);

    return 0;
}
```

## P4: Pointers Basics - Pointer is an integer

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra que el puntero es una variable que almacena un valor entero (es decir, la dirección).

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
```

```
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t081_ch3_pointer_is_an_int.c
 *   Title          : Pointer Basics - Pointer is an integer
 *   Description    : This example shows that the pointer is a variable which stores an
 *                    integer value (i.e the address).
 *-----------------------------------------------------------------------------------------*/
#include
int main()
{
    int num;
    int *ptr;
    /* Store a value in a integer variable */
    num = 5;
    /* Store a value in a pointer variable */
    ptr = 5;
    /*
    * There is no difference in both the above statements, They just hold integer numbers!.
    * On this sense pointers always integral values (Address can never be real number, at least not
    * in computers ;) )
    * But most important point is that, the width of the address could be bigger that it might not fit
    * in a variable of type int!!
    * Width of the address depends on the addressing capability of the system.
    */
    printf("num = %dn", num);
    printf("ptr = %dn", ptr);
    /* Check for exception (Size might differ) */
    printf("sizeof int = %un", sizeof(int));
    printf("sizeof long = %un", sizeof(long));
    printf("sizeof pointer = %un", sizeof(int *));
    return 0;
}
```

## P5: Pointers Basics - Why types to Pointers? - Part 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra por qué los tipos asociados a punteros siempre contienen valores enteros, como se vio en el ejemplo anterior.

Source Code:

```
/*-----------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t082_ch3_pointer_types.c
 *   Title          : Pointer Basics - Why types to Pointers?
 *   Description    : This example shows why the types attached to pointer when they always
 *                    hold integral values as seen in the previous example.
 *-----------------------------------------------------------------------------------------*/
#include
int main()
{
    double d = 2.5;
    char ch = 'A';
    int *iptr;
    char *cptr;
    double *dptr;
```

```
    printf("Sizeof *iptr = %un", sizeof(*iptr));
    printf("Sizeof *cptr = %un", sizeof(*cptr));
    printf("Sizeof *dptr = %un", sizeof(*dptr));
    /* Store address of char ch */
    cptr = &ch;
    /* Store address of double d  */
    dptr = &d;
    /* De-reference char pointer (fetches a char - 1 byte) */
    printf("*cptr = %cn", *cptr);
    /* De-reference double pointer (fetches a double - 8 bytes) */
    printf("*dptr = %fn", *dptr);
    /*
    * From the above statements we can conclude the types to pointers are
    * required to access the value pointed by the pointer, without which
    * it would not know how many bytes to read or write from / to the location
    * it is pointing to.
    */
 return 0;
}
```

## P6: Pointers Basics - Why types to Pointers? - Part 2

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra que el tamaño de todos los punteros sigue siendo el mismo que el tamaño del bus de direcciones.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t083_ch3_pointer_size.c
 *   Title          : Pointer Basics - Why types to Pointers?
 *   Description    : This example shows all the pointers size remains the same as address
 *                    bus size.
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
 char *cptr;
 int *iptr;

 if (sizeof(cptr) == sizeof(iptr))
 {
  printf("Size of all pointers are equaln");
 }
 else
 {
  printf("No they are not equaln");
 }

 if (sizeof(char *) == sizeof(long long *))
 {
  printf("Size of all pointers are equaln");
```

```
 }
 else
 {
  printf("No they are not equaln");
 }

 return 0;
}
```

## P7: Pointers Basics - Cross accessing different pointer types

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo muestra lo que sucede si se realiza un acceso cruzado a distintos tipos de punteros.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t084_ch3_pointer_types.c
 *   Title          : Pointer Basics - Cross accessing different pointer types
 *   Description    : This example shows what happens if cross access different pointers types
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
    int num = 0x12345678;
    int *ptr = #
    char *cptr = (char *)#

    int int_val;
    char char_val;

    int_val = *ptr;
    char_val = *cptr;

    printf("*ptr = %x, int_val = %xn", *ptr, int_val);
    printf("*cptr = %x, char_val = %xn", *cptr, char_val);

    *cptr = 0x55;
    printf("*cptr = %x, char_val = %xn", *cptr & 0xFF, char_val);
    printf("*ptr = %x, int_val = %xn", *ptr, int_val);

    return 0;
}
```

## P8: Pointers Basics - Array Interpretation

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros.

Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo demuestra la diferente interpretación de una matriz por parte del compilador.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Fri 14 April 2017 16:00:04 IST
 *   File           : c_t201_preprocessor_macros.c
 *   Title          : Preprocessors - Macros - Stringification - Example 1
 *   Description    : One of the step performed before compilation. Is a text substitution tool
 *                    and it instructs the compiler to do required pre-processing before the
 *                    actual compilation.
 *                    Instructions given to preprocessor are called preprocessor directives and
 *                    they begin with ‘‘#’’ symbol.
 *                    Few advantages of using preprocessor directives would be,
 *                     - Easy Development
 *                     - Readability
 *                     - Portability
 *
 *                  This example shows how a macro argument can be converted into a string.
 *-------------------------------------------------------------------------------------------*/

#include

#define WARN_IF(EXP)
do
{
 x--;
 if (EXP)
 {
  fprintf(stderr, "Warning: " #EXP "n");
 }
} while (x)

int main()
{
 int x = 5;

 WARN_IF (x == 0);

 return 0;
}
```

## P9: Pointers Basics - Array Arithmetic - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo demuestra operaciones aritméticas con punteros.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Fri 14 April 2017 16:00:04 IST
 *   File           : c_t202_preprocessor_macros.c
```

```
 *   Title          : Preprocessors - Macros - Stringification - Example 2
 *   Description    : This example shows how a macro argument can be converted into a string.
 *-------------------------------------------------------------------------------------------*/

#include

#define PRINT(expr) printf(#expr " is %dn", expr)

int main()
{
    int x, y = 5, z = 10, a = 50, b = 12, c = 100;

    PRINT(x = y + c * (a / z) % y);

 return 0;
}
```

## P10: Pointers Basics - Array Arithmetic - Example 2

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo demuestra operaciones aritméticas con punteros.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Fri 14 April 2017 16:00:04 IST
 *   File           : c_t203_preprocessor_macros.c
 *   Title          : Preprocessors - Macros - Concatination
 *   Description    : This example shows how a macro argument can be used to concatinated.
 *-------------------------------------------------------------------------------------------*/

#include

#define CAT(x) (x##_val)

int main()
{
 int int_val = 4;
 float float_val = 2.54;

 printf("int val = %dn", CAT(int));
 printf("float val = %fn", CAT(float));

 return 0;
}
```

## P11: Pointers Basics - Array Arithmetic - Example 3

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que

tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo demuestra operaciones aritméticas con punteros.

Source Code:

```
/*------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t088_ch3_array_interpretations.c
 *   Title          : Pointer Basics - Array Arithmetic - Example 3
 *   Description    : This example demonstrates arithmetic operations on pointers.
 *------------------------------------------------------------------------------------------*/

#include

int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int *ptr;
    int i;

    /*
     * This is the second interpretation by the compiler as a pointer to the first element
     * of an array
     * a is a pointer to the first element
     */
    ptr = a;

    for (i = 0; i < 5; i++)
    {
        /*
         * The pointer is incremented every time the loop is run accessing each element of
         * an array its pointing to.
         * Note: The post increment happens first and the dereferencing next. Still the
         * pointer variable will be holding the older value which get dereferenced
         */
        printf("%d ", *ptr++);
    }
    printf("n");

    return 0;
}
```

## P12: Pointers Basics - Pointer vs Arrays Example 1 - Part 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo demuestra la principal diferencia entre punteros y matrices en la forma en que los manejamos.

Source Code:

```
/*------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
```

```
 *   File           : c_t089_ch3_pointers_vs_array.c
 *   Title          : Pointer Basics - Pointer vs Arrays Example 1 - Part 1
 *   Description    : This example demonstrates the major difference between pointers and
 *                    arrays on we handle it.
 *-----------------------------------------------------------------------------------------*/

#include

int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int *ptr;
    int i;

    /*
     * This is the second interpretation by the compiler as a pointer to the first element
     * of an array
     * a is a pointer to the first element
     */
    ptr = a;

    for (i = 0; i < 5; i++)
    {
        /* Allowed. As explained in the previous example */
        printf("%d ", *ptr++);
    }
    printf("n");

    for (i = 0; i < 5; i++)
    {
        /* Not allowed. The array name represents a constant pointer */
        printf("%d ", *a++);
    }
    printf("n");

    return 0;
}
```

## P13: Pointers Basics - Pointer vs Arrays Example 1 - Part 2

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. Este ejemplo demuestra la principal diferencia entre punteros y matrices en la forma en que los manejamos.

Source Code:

```
/*-----------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Thu 23 Mar 2016 16:00:04 IST
 *   File           : c_t090_ch3_pointers_vs_array.c
 *   Title          : Pointer Basics - Pointer vs Arrays Example 1 - Part 2
 *   Description    : This example demonstrates the major difference between pointers and
 *                    arrays on we handle it.
 *-----------------------------------------------------------------------------------------*/

#include
```

```
int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int *ptr;
    int i;

    /*
     * This is the second interpretation by the compiler as a pointer to the first element
     * of an array
     * a is a pointer to the first element
     */
    ptr = a;

    for (i = 0; i < 5; i++)
    {
        /* Allowed. As explained in the previous example */
        printf("%d ", *ptr++);
    }
    printf("n");

    for (i = 0; i < 5; i++)
    {
        /* Allowed. Here we are indexing the elements from the base address */
        printf("%d ", *(a + i));
    }
    printf("n");

    return 0;
}
```

# Chapter 10 : Pointers Part 2

## P1: NUII Pointers - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra el uso de punteros NULL.

Source Code:

```
/*-----------------------------------------------------------------------------------------
 *   Author          : Emertxe (https://www.emertxe.com)
 *   Date            : Wed 12 April 2017 16:00:04 IST
 *   File            : c_161_ch5_pointer_null.c
 *   Title           : NUll Pointers - Example 1
 *   Description     : The C basic data types allows us to access a limited and fixed number of the
 *                     memory bytes. In order to increase the flexibility to access a range of
 *                     memory bytes C supports pointers.
 *                     Any variable defined as a pointer will hold an address of memory location.
 *                     Have to be careful with dealing the address ranges, else would lead to
 *                     illegal memory access.
 *                     This example shows use of NULL pointers
 *-----------------------------------------------------------------------------------------*/
```

```
#include
#include
#include

int main()
{
    /*
     * Note here we have not assigned a valid address to the pointer yet.
     * This is sometimes referred as wild pointers
     */
    int *iptr;

    /* Accessing the uninitialized pointer could have undefined behavior */
    *iptr = 50;
    /* You might get 50 on screen */
    printf("%dn", *iptr);

    /*
     * If you still don't have any valid address to be stored in a pointer, better
     * initialized that pointer to a value which would give a predicted value all
     * the time if accessed unknowingly
     * This can be achieved by initializing a pointer to NULL (OS dependent reserved value)
     * when accessed gives you guaranteed segmentation violation error
     */
    iptr = NULL;

    /* Will lead to segmentation fault */
    *iptr = 50;
    printf("%dn", *iptr);

    return 0;
}
```

## P2: Void Pointers - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra el uso de punteros void.

Source Code:

```
/*-----------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_162_ch5_void_pointers.c
 *   Title          : Void Pointers - Example 1
 *   Description    : This example shows use of void pointers
 *-----------------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 0x31323334;
    /*
    * The advantage of the void pointer is that we can make to point to any memory location where
```

```
    * we don't bother what is stored at the location
    */
    void *vptr = &x;

    /*
    * As mention above the void pointer will be pointing a memory location containing just data.
    * Now while accessing the data the compiler expects us to tell how much we need
    * This has to be done by type casting the void pointer
    */
    printf("%cn", *(char *)(vptr + 0));
    printf("%cn", *(char *)(vptr + 1));
    printf("%cn", *(char *)(vptr + 2));
    printf("%cn", *(char *)(vptr + 3));

    printf("%hxn", *(short *)(vptr + 2));

    printf("%xn", *(int *)(vptr + 0));

    return 0;
}
```

## P3: Dynamic Memory Allocation - malloc - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo utilizar malloc.

Source Code:

```
/*-----------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_163_ch5_pointers_malloc.c
 *   Title          : Dynamic Memory Allocation - malloc - Example 1
 *   Description    : This example shows how to use malloc
 *----------------------------------------------------------------------------------------------*/

#include
#include

int main()
{
    int *ptr;
    int i;

    /* Request 5 blocks of memory of size integer */
    ptr = (int *) malloc(5 * sizeof(int));

    if (ptr == NULL)
    {
          fprintf(stderr, "malloc: Memory not allocatedn");

          return 1;
    }

    ptr[0] = 25;
```

```
    ptr[1] = 50;
    ptr[2] = 75;
    ptr[3] = 100;
    ptr[4] = 125;

 for (i = 0; i < 5; i++)
 {
  printf("%dn", i[ptr]);
 }

    /* Make sure you free the allocated memory to avoid memory leak */
    free(ptr);

    return 0;
}
```

## P4: Dynamic Memory Allocation - calloc - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo utilizar calloc.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_164_ch5_pointers_calloc.c
 *   Title          : Dynamic Memory Allocation - calloc - Example 1
 *   Description    : This example shows how to use calloc
 *-------------------------------------------------------------------------------------------*/

#include
#include

int main()
{
    int *ptr;
    int i;

    /* Request 10 blocks of memory of size integer initialized with 0 */
    ptr = (int *) calloc(10, sizeof(int));

    if (ptr == NULL)
    {
          fprintf(stderr, "calloc: Memory not allocatedn");

          return 1;
    }

    *(ptr + 0) = 25;
    *(ptr + 1) = 50;
    *(ptr + 2) = 75;
    *(ptr + 3) = 100;
    *(ptr + 4) = 125;
```

```
    /*
     * Not that the loop runs for 10 times, but we have assigned only 5 values
     * which will be initialized to 0
     */
 for (i = 0; i < 5; i++)
 {
  printf("%dn", i[ptr]);
 }

    /* Make sure you free the allocated memory to avoid memory leak */
    free(ptr);

    return 0;
}
```

## P5: Dynamic Memory Allocation - realloc - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria. La asignación dinámica de memoria nos ayuda a obtener la memoria del montón en tiempo de ejecución. La memoria asignada se puede ampliar o reducir.

Este ejemplo muestra cómo utilizar realloc.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_165_ch5_pointers_realloc.c
 *   Title          : Dynamic Memory Allocation - realloc - Example 1
 *   Description    : This example shows how to use realloc
 *-------------------------------------------------------------------------------------------*/

#include
#include

int main()
{
    int *ptr;
    int *new_ptr;

    /* Allocate 5000 * 4 (32 bit system) bytes of memory */
    ptr = (int *) malloc(5000 * sizeof(int));

    if (ptr == NULL)
    {
          fprintf(stderr, "calloc: Memory not allocatedn");

          return 1;
    }

    /*
     * Just reallocate big chunk of memory
     * If the kernel doesn't find enough space in existing block then it would
     * give a new memory block if possible else would fail
     */
    new_ptr = realloc(ptr, 0x80000000);
```

```
    if (new_ptr == NULL)
    {
       fprintf(stderr, "malloc: Memory not allocatedn");
    }
    else
    {
      ptr = new_ptr;
    }

    printf("Pointing to region starting from %pn", ptr);

    /* Make sure you free the allocated memory to avoid memory leak */
    free(ptr);

    return 0;
}
```

## P6: Const Pointers - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo utilizar punteros constantes.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_166_ch5_pointers_const.c
 *   Title          : Const Pointers- Example 1
 *   Description    : This example shows how to use const pointers
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 100;
    /* *ptr is constant */
    const int *ptr = &x;

    /* This is allowed */
    ptr++;

    /* Is not OK */
    *ptr = 5;

    return 0;
}
```

## P7: Const Pointers - Example 2

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo utilizar punteros constantes.
Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_167_ch5_pointers_const.c
 *   Title          : Const Pointers- Example 2
 *   Description    : This example shows how to use const pointers
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 100;
    /* *ptr is constant */
    int * const ptr = &x;

    /* This is not allowed */
    ptr++;

    /* This is allowed */
    *ptr = 5;

    return 0;
}
```

## P8: Const Pointers - Example 3

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo utilizar punteros constantes.
Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_168_ch5_pointers_const.c
 *   Title          : Const Pointers- Example 3
 *   Description    : This example shows how to use const pointers
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
    int x = 100;
```

```
    /* Both pointer and the value it is pointing are constants */
    const int * const ptr = &x;

    /* This is not allowed */
    ptr++;

    /* This is not allowed */
    *ptr = 5;

    return 0;
}
```

# Chapter 11 : Pointers Part 3

## P1: Multilevel Pointers - Example 1

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo definir punteros multinivel.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_161_ch5_pointer_multi.c
 *   Title          : Multilevel Pointers - Example 1
 *   Description    : The C basic data types allows us to access a limited and fixed number of the
 *                    memory bytes. In order to increase the flexibility to access a range of
 *                    memory bytes C supports pointers.
 *                    Any variable defined as a pointer will hold an address of memory location.
 *                    Have to be careful with dealing the address ranges, else would lead to
 *                    illegal memory access.
 *                    This example shows how to define multilevel pointers
 *-------------------------------------------------------------------------------------------*/

#include

int main()
{
 int num = 10;
 /* ptr1 points to num */
 int *ptr1 = #
 /* ptr2 points to ptr1 */
 int **ptr2 = &ptr1;
 /* ptr3 points to ptr2 */
 int ***ptr3 = &ptr2;

 printf("%pn", ptr3);
 printf("%pn", *ptr3);
 printf("%pn", **ptr3);
 printf("%dn", ***ptr3);

 return 0;
}
```

## P2: Multilevel Pointers - Example 2

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo definir punteros multinivel y pasarlos a una función.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_170_ch5_pointer_multi.c
 *   Title          : Multilevel Pointers - Example 2
 *   Description    : This example shows how to define multilevel pointers
 *-------------------------------------------------------------------------------------------*/

#include

void foo(int **pptr)
{
    **pptr = 50;
}

int main()
{
    int x = 10;
    int *ptr = &x;

    foo(&ptr);

    printf("x = %dn", x);

    return 0;
}
```

## P3: Multilevel Pointers - Example 3

Resumen:

Los tipos de datos básicos de C nos permiten acceder a un número limitado y fijo de bytes de memoria. Para aumentar la flexibilidad de acceso a un rango de bytes de memoria, C admite punteros. Cualquier variable definida como puntero contendrá una dirección de ubicación de memoria. Hay que tener cuidado al tratar los rangos de direcciones, ya que de lo contrario se produciría un acceso ilegal a la memoria.

Este ejemplo muestra cómo definir punteros multinivel y pasarlos a una función y asignarle memoria.

Source Code:

```
/*-------------------------------------------------------------------------------------------
 *   Author         : Emertxe (https://www.emertxe.com)
 *   Date           : Wed 12 April 2017 16:00:04 IST
 *   File           : c_171_ch5_pointer_multi.c
 *   Title          : Multilevel Pointers - Example 3
 *   Description    : This example shows how to define multilevel pointers and passing to a function
 *                    and allocation of memory in it.
 *-------------------------------------------------------------------------------------------*/
```

```
#include
#include

void alloc_mem(int **pptr, int nmemb)
{
    *pptr = (int *)malloc(nmemb * sizeof(int));
}

int main()
{
    int *ptr = NULL;

    alloc_mem(&ptr, 5);

    if (ptr == NULL)
    {
        exit(1);
    }

    printf("Allocated memoryn");

    if (ptr != NULL)
    {
        free(ptr);
        ptr = NULL;
    }

    return 0;
}
```