

A close-up, slightly blurred photograph of a laptop. The screen shows a code editor with syntax-highlighted code, likely for a web application. The code includes HTML tags like <Link>, <div>, <Preview>, <Editor>, and <button>, along with JavaScript logic using {code} and evalInContext. The keyboard is visible in the foreground, showing keys like 'P', 'R', 'E', 'S', 'T', 'N', 'T', 'S', 'P', 'A', 'C', 'E', 'B', 'A', 'C', 'K', 'S', 'P', 'A', 'C', 'E', 'B', 'A', 'C', 'K', 'S', 'P', 'A', 'C', 'E'. A dark semi-transparent banner is overlaid on the left side of the image, containing the text 'AMIT LEARNING' and 'SESSION 4 POINTER'. The 'AMIT' logo is in the bottom right corner.

# AMIT LEARNING

SESSION 4 POINTER

## CONTENTS

1	WHAT IS POINTER	1
1.1	Introduction to Pointers.	1
1.2	Type of machine	
1.3	Pointer Arithmetic.	1
1.4	Passing pointer to function.	1
1.5	What is constant and constant with pointer	1
1.6	Dangling, Void , Null and Wild Pointers.	1
1.7	Double pointer in c.	1
1.8	How to declare a pointer to a function.	1
1.9	What are near, far and huge pointers?	1

1

Introduction to pointer

1.1 Introduction to pointer

➤ Pointers store address of variables or a memory location.

```
// General syntax
datatype *var_name;

// An example pointer "ptr" that holds
// address of an integer variable or holds
// address of a memory whose value(s) can
// be accessed as integer values through "ptr"
int *ptr;
```

## How pointer works in C

```
int var = 10;                      →      var
                                         10
                                         #2008

int *ptr = &var;                   →
*ptr = 20;                           →      20
```

- To use pointers in C, we must understand below two operators.
- To access address of a variable to a pointer, we use the unary operator **&** (ampersand) that returns the address of that variable. For example **&x** gives us address of variable x.
  - To declare a pointer variable: When a pointer variable is declared in C there must be a **\*** before its name.
  - To access the value stored in the address we use the unary operator **(\*)** that returns the value of the variable located at the address specified by its operand. This is also called **Dereferencing**.

1

## Introduction to pointer

## 1.1 Introduction to pointer

## CODE


```
// The output of this program can be different
// in different runs. Note that the program
// prints address of a variable and a variable
// can be assigned different address in different
// runs.
#include <stdio.h>

int main()
{
    int x;

    // Prints address of x
    printf("%p", &x);

    return 0;
}
```

## OUTPUT



000000000061FE1C

## DESCRIPTION

- In this example print address of variable.



1

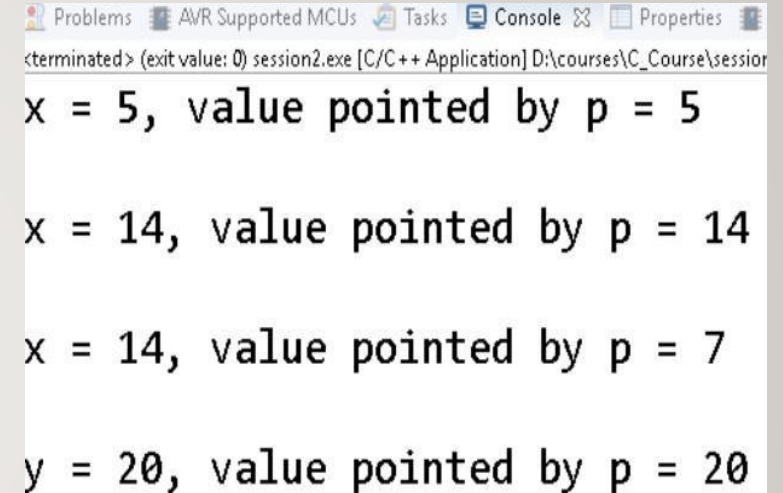
## Introduction to pointer

## 1.1 Introduction to pointer

## CODE

```
2 #include <stdio.h>
3 int main(int argc, char**argv) {
4     int x = 5;
5     int y = 7;
6     int* p;
7     p = &x;
8     printf("x = %d, value pointed by p = %d\r\n", x, *p);
9     *p = 14;
10    printf("x = %d, value pointed by p = %d\r\n", x, *p);
11    p = &y;
12    printf("x = %d, value pointed by p = %d\r\n", x, *p);
13    *p = 20;
14    printf("y = %d, value pointed by p = %d\r\n", y, *p);
15    p = 0;
16    /*p = 15; //Wrong and the program will crash here
17    return 0 ;
18 }
```

## OUTPUT



```
<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
```

## DESCRIPTION

- In this example discuss about pointer .

1

## Introduction to pointer

## 1.1 Introduction to pointer

## CODE

```
int main()
{
    int var = 10;

    // 1) Since there is * in declaration, ptr
    // becomes a pointer variable (a variable
    // that stores address of another variable)
    // 2) Since there is int before *, ptr is
    // pointer to an integer type variable
    int *ptr;

    // & operator before x is used to get address
    // of x. The address of x is assigned to ptr.
    ptr = &var;
    printf("address ptr:%p\n", ptr);
    printf("*ptr:%d\n", *ptr);
    printf("*var:%d\n", var);

    *ptr=20; // We can also use ptr as lvalue (Left hand
    // side of assignment)
    // Value at address is now 20
    printf("*ptr:%d\n", *ptr);
    printf("*var:%d\n", var);
    return 0;
}
```

## OUTPUT

```
address ptr:000000000061FE14
*ptr:10
*var:10
*ptr:20
*var:20
```

## DESCRIPTION

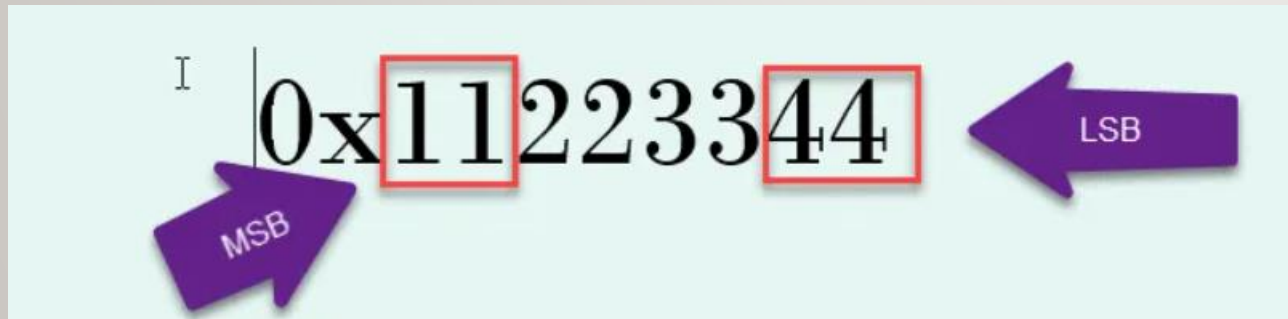
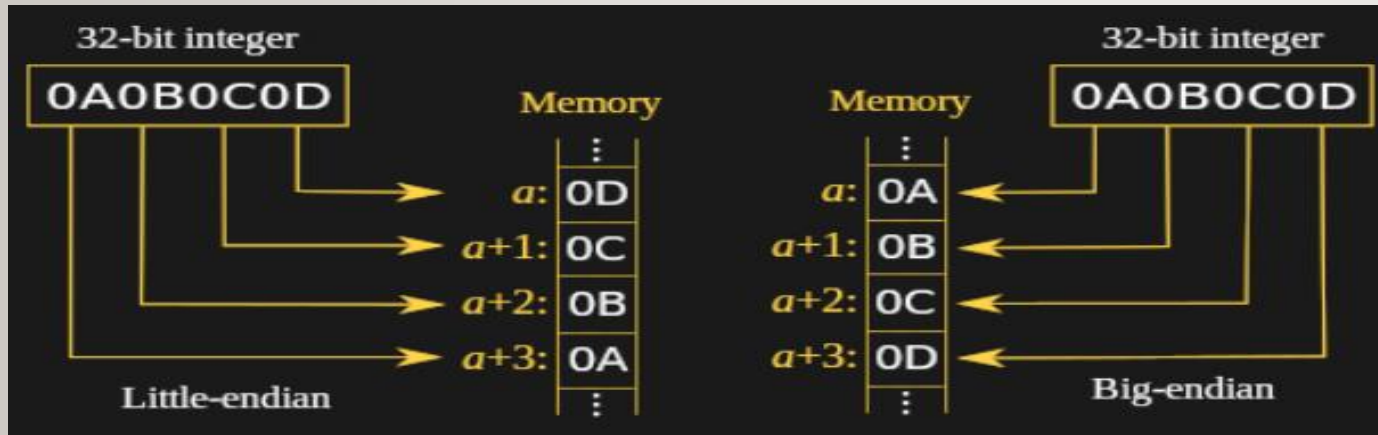
- In this example print content of pointer and dereference of pointer.

## 1 Type of machine

### 1.2 Type of machine

➤ Type of machine divided to two category

1. Little endian: the LSB byte store first (means at lower address).
2. Big endian: the MSB byte store first (means at lower address).



#### Little Endian

- Intel x86 and x86-64 series
- Zilog Z80 (including Z180 and eZ80)
- MOS Technology 6502

#### Big Endian

- Motorola 68000 series
- Xilinx Microblaze, SuperH,
- IBM z/Architecture, Atmel AVR32.



1

## Type of machine

## 1.2 Type of machine

## CODE

```
int main()
{
    int x=0x11223344;
    char *ptr=(char*)&x;
    if(*ptr==0x44)
    {
        printf("this is little endian");
    }
    else
    {
        printf("this is big endian");
    }
    return 0;
}
```

## OUTPUT

```
this is little endian
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

## DESCRIPTION

- In this example check my machine is little endian and big endian.

1

## Pointer Arithmetic

## 1.3 Pointer Arithmetic

## CODE

```
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 }
```

## OUTPUT

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

## DESCRIPTION

- In this example used arithmetic pointer in c .

1

## Pointer Arithmetic

## 1.3 Pointer Arithmetic

## CODE

```
int main(void)
{
    int arr=0x412345;
    char *p = &arr;
    ++*p;
    printf("%x\n", *p);
    *p++;
    printf("%x\n", *p);
    *++p;
    printf("%x", *p);

    return 0;
}
```

## OUTPUT

```
*P:46
*p:23
*p:41
```

## DESCRIPTION

- In this example used arithmetic pointer in c .

## 1 Passing pointer to function

### 1.4 Passing pointer to function.

➤ 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 (Calling by Reference or Pointer)**

The parameter pointer (reference) 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.

## 1 Passing pointer to function

### 1.4 Passing pointer to function

#### CODE

```
void salaryhike(int *var, int *b);

int main()
{
    int salary=0, bonus=0;
    printf("Enter the employee current salary:");
    scanf("%d", &salary);
    printf("Enter bonus:");
    scanf("%d", &bonus);
    salaryhike(&salary, &bonus);
    printf("salary:%d \n bouns:%d", salary, bonus);
    return 0;
}

void salaryhike(int *var, int* b)
{
    *var = *var+1;
    *b=*b+1;
    printf("*var:%d\n*p:%d\n", *var, *b);
}
```

#### OUTPUT

```
Enter the employee current salary:1
Enter bonus:2
*var:2
*p:3
salary:2
bouns:3
```

#### DESCRIPTION

- In this example we used call by reference and arguments passed to function is changed in another function.



# 1 Passing pointer to function

## 1.4 Passing pointer to function

### CODE

```
void salaryhike(int var, int b);

int main()
{
    int salary=0, bonus=0;
    printf("Enter the employee current salary:");
    scanf("%d", &salary);
    printf("Enter bonus:");
    scanf("%d", &bonus);
    salaryhike(salary, bonus);
    printf("salary:%d \n bouns:%d", salary, bonus);
    return 0;
}

void salaryhike(int var, int b)
{
    var = var+1;
    b=b+1;
    printf("*var:%d\n*p:%d\n", var, b);
}
```

### OUTPUT

```
Enter the employee current salary:1
Enter bonus:2
*var:2
*p:3
salary:1
bouns:2
```

### DESCRIPTION

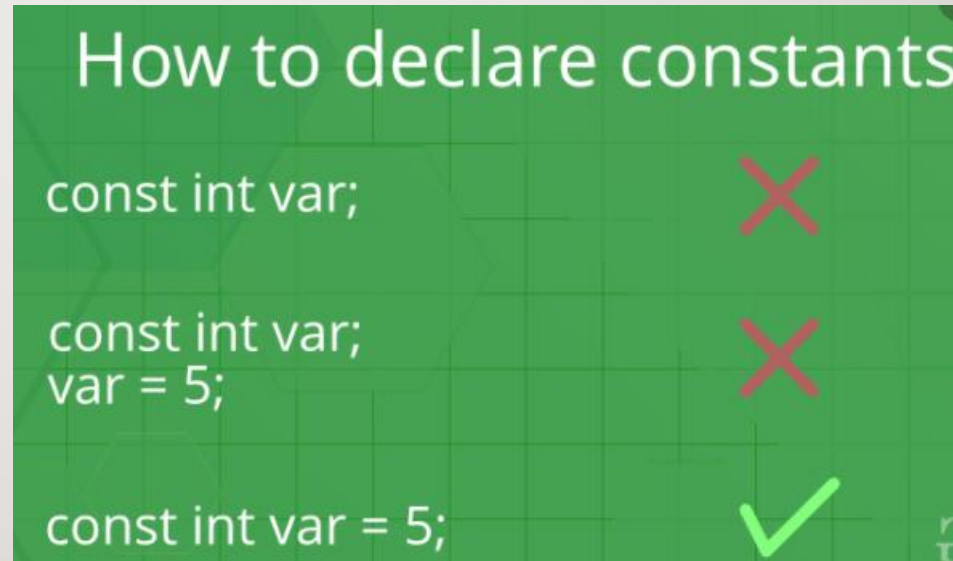
- In this example we used call by value and arguments passed to function isn't changed in another function.

## 1 What is constant and constant with pointer

### 1.5 What is constant and constant with pointer

- Constants refer to fixed values that the program may not alter during its execution and put in .rodata
- Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*.
- There are enumeration constants as well.
- String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals.
- Defining Constants
  - There are two simple ways in C to define constants
    - Using **#define** preprocessor.
    - Using **const** keyword. Any variable cannot be hacked by pointer put in .rodata, Any variable can be hacked by pointer put in stack
    - Syntax:

```
const type variable = value;
```



## 1 What is constant and constant with pointer

## 1.5 What is constant and constant with pointer

## CODE

```
#include<stdio.h>

int main()
{
    const int x=4;
    int*p=&x;
    *p+=1;
    printf("%d",*p);
    return 0;
}
```

## OUTPUT

```
5
Process returned 0 (0x0)   execution time : 0.065 s
Press any key to continue.
```

## DESCRIPTION

- In this example variable x is constant and hack by pointer so put on stack

## 1 What is constant and constant with pointer

## 1.5 What is constant and constant with pointer

## CODE

```
#include<stdio.h>
const int x=4;
int main()
{
    int*p=&x;
    *p+=1;
    printf("%d", *p);
    return 0;
}
```

## OUTPUT

- Run Time Error.

## DESCRIPTION

- In this example variable x is constant and global and cannot be hacked by pointer so it is put in .rodata

## 1 What is constant and constant with pointer

### 1.5 What is constant and constant with pointer

➤ **Difference between `const char *p`, `char * const p` and `const char * const p`?**

**1) `const char *ptr` :** This is a pointer to a constant character. **You cannot change the value pointed by ptr,**

- **but you can change the pointer itself.** “`const char *`” is a (non-const) pointer to a const char.
- **NOTE:** There is no difference between **`const char *p` and `char const *p`** as both are pointer to a const char and position of ‘\*’(asterik) is also same.

**2) `char *const ptr` :** This is a constant pointer to non-constant character. **You cannot change the pointer p, but can change the value pointed by ptr.**

- **NOTE:** Pointer always points to same address, only the value at the location is changed.

**3) `const char * const ptr` :** This is a constant pointer to constant character. **You can neither change the value pointed by ptr nor the pointer ptr.**

**NOTE:** `char const * const ptr` is same as `const char *const ptr`.



## 1 What is constant and constant with pointer

## 1.5 What is constant and constant with pointer

## CODE

```
// C program to illustrate
// char const *p
#include<stdio.h>
#include<stdlib.h>

int main()
{
    char a = 'A', b = 'B';
    const char *ptr = &a;

    /*ptr = b; illegal statement (assignment of read-only location *ptr)

    // ptr can be changed
    printf( "value pointed to by ptr: %c\n", *ptr);
    ptr = &b;
    printf( "value pointed to by ptr: %c\n", *ptr);
}
```

## OUTPUT

value pointed to by ptr:A  
value pointed to by ptr:B

## DESCRIPTION

- In this example ptr is non constant address and constant value.

## 1 What is constant and constant with pointer

## 1.5 What is constant and constant with pointer

## CODE

```
int main()
{
    char a = 'A', b = 'B';
    char *const ptr = &a;
    printf( "Value pointed to by ptr: %c\n", *ptr);
    printf( "Address ptr is pointing to: %d\n\n", ptr);

    //ptr = &b; illegal statement (assignment of read-only variable ptr)

    // changing the value at the address ptr is pointing to
    *ptr = b;
    printf( "Value pointed to by ptr: %c\n", *ptr);
    printf( "Address ptr is pointing to: %d\n", ptr);
}
```

## OUTPUT

Value pointed to by ptr: A  
Address ptr is pointing to: -1443150762

Value pointed to by ptr: B  
Address ptr is pointing to: -1443150762

## DESCRIPTION

- In this example ptr is constant address and non constant value.

## 1 What is constant and constant with pointer

## 1.5 What is constant and constant with pointer

## CODE

```
// C program to illustrate
//const char * const ptr
#include<stdio.h>
#include<stdlib.h>

int main()
{
    char a = 'A', b = 'B';
    const char *const ptr = &a;

    printf( "Value pointed to by ptr: %c\n", *ptr);
    printf( "Address ptr is pointing to: %d\n\n", ptr);

    // ptr = &b; illegal statement (assignment of read-only variable ptr)
    // *ptr = b; illegal statement (assignment of read-only location *ptr)
}
```

## OUTPUT

```
Value pointed to by ptr: A
Address ptr is pointing to: -255095482
```

## DESCRIPTION

- In this example ptr is constant address and constant value.

## 1 Dangling, Void , Null and Wild Pointers

### 1.6 Dangling, Void , Null and Wild Pointers.

#### ➤ Dangling pointer:

- A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.
- **Function Call, Variable goes out of scope.**

#### ➤ Void pointer:

- Void pointer is a specific pointer type – void \* – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer.

#### Important Points

- void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer.
- Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size. The C standard doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of void is 1

#### ➤ NULL Pointer:

- NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

#### ▪ Important Points

- **NULL vs Uninitialized pointer** – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
- **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

## 1 Dangling, Void , Null and Wild Pointers

### 1.6 Dangling, Void , Null and Wild Pointers.

#### ➤ wild pointer:

- A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.



## 1 Dangling, Void , Null and Wild Pointers

## 1.6 Dangling, Void , Null and Wild Pointers.

## CODE

```
int *fun();
int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not
    // valid anymore
    printf("%d", *p);
    return 0;
}
int *fun()
{
    // x is local variable and goes out of
    // scope after an execution of fun() is
    // over.
    int x = 5;

    return &x;
}
```

## OUTPUT

- Run time error

## DESCRIPTION

- In this example dangling pointer return pointer to integer and dereference pointer is undefined behavior.

# 1 Dangling, Void , Null and Wild Pointers

## 1.6 Dangling, Void , Null and Wild Pointers.

### CODE

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

    {
        int ch=6;
        ptr = &ch;
    }

    // Here ptr is dangling pointer
    printf("%d", *ptr);

    return 0;
}
```

### OUTPUT

▪ No error

# 1 Dangling, Void , Null and Wild Pointers

## 1.6 Dangling, Void , Null and Wild Pointers.

### CODE

```
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

### OUTPUT

Compiler Error: 'void\*' is not a pointer-to-object type

### DESCRIPTION

- In this example return pointer to integer and dereference pointer is compiler error.

## 1 Dangling, Void , Null and Wild Pointers

## 1.6 Dangling, Void , Null and Wild Pointers.

## CODE

```
int main()
{
    int x = 4;
    float y = 5.5;

    //A void pointer
    void *ptr;
    ptr = &x;

    // (int*)ptr - does type casting of void
    // *((int*)ptr) dereferences the typecasted
    // void pointer variable.
    printf("Integer variable is = %d", *( (int*) ptr) );

    // void pointer is now float
    ptr = &y;
    printf("\nFloat variable is= %f", *( (float*) ptr) );

    return 0;
}
```

## OUTPUT

```
Integer variable is = 4
Float variable is= 5.500000
```

## DESCRIPTION

- In this example pointer to void.

1

## Double pointer in c

## 1.7 Double pointer in c

➤ **How to declare a pointer to pointer in C?**

- Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '\*' before the name of pointer.
- Syntax:

```
int **ptr;    // declaring double pointers
```





1

## Double pointer in c

### 1.7 Double pointer in c

#### CODE

```
int main()
{
    int var = 789;

    // pointer for var
    int *ptr2;

    // double pointer for ptr2
    int **ptr1;

    // storing address of var in ptr2
    ptr2 = &var;

    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;

    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var );
    printf("Value of var using single pointer = %d\n", *ptr2 );
    printf("Value of var using double pointer = %d\n", **ptr1);

    return 0;
}
```

#### OUTPUT

```
Value of var = 789
Value of var using single pointer = 789
Value of var using double pointer = 789
```

#### DESCRIPTION

- In this example pointer to pointer.

## 1 How to declare a pointer to a function

### 1.8 How to declare a pointer to a function.

#### ➤ How to declare a pointer to a function?

- 1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- 2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- 3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing
- 4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.
- 5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.
- 6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

# 1 How to declare a pointer to a function

## 1.8 How to declare a pointer to a function

### CODE

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

### OUTPUT

Value of a is 10

### DESCRIPTION

- In this example call function by pointer to function

# 1 How to declare a pointer to a function

## 1.8 How to declare a pointer to a function

### CODE

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

### OUTPUT

Value of a is 10

### DESCRIPTION

- In this example call function by pointer to function

## 1 How to declare a pointer to a function

### 1.8 How to declare a pointer to a function

#### CODE

```
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

#### OUTPUT

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply
2
Multiplication is 150
```

#### DESCRIPTION

- In this example call function by pointer to function using array of pointer to function.

# 1 How to declare a pointer to a function

## 1.8 How to declare a pointer to a function

### CODE

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

### OUTPUT

```
Fun1
Fun2
```

### DESCRIPTION

- In this example pass pointer function to function.



## 1 What are near, far and huge pointers

### 1.9 What are near, far and huge pointers



What are near, far and huge pointers?

These are some old concepts used in 16 bit intel architectures in the days of MS DOS, not much useful anymore.

- **Near pointer** is used to store 16 bit addresses means within current segment on a 16 bit machine. The limitation is that we can only access 64kb of data at a time.
- **A far pointer** is typically 32 bit that can access memory outside current segment. To use this, compiler allocates a segment register to store segment address, then another register to store offset within current segment.  
Like far pointer.
- **huge pointer** is also typically 32 bit and can access outside segment. In case of far pointers, a segment is fixed.