UNIT-III

FUNCTIONS AND POINTERS

FUNCTIONS

A function is itself a block of code which can solve simple or complex task/calculations.

A function performs calculations on the data provided to it is called "parameter" or "argument".

A function always returns single value result.

Types of function:

1. Built in functions(Library functions)

a.) Inputting Functions.

b.) Outputting functions.

2. User defined functions.

a.) fact();

b.) sum();

Parts of a function:

1. Function declaration/Prototype/Syntax.

2. Function Calling.

3. Function Definition.

1.)Function Declaration:

Syntax: <return type > <function name>(<type of argument>)

The declaration of function name, its argument and return type is called function declaration.

2.) Function Calling:

The process of calling a function for processing is called function calling.

Syntax: <var_name>=<function_name>(<list of arguments>).

3.) Function defination:

Scanned with CamScanner

The process of writing a code for performing any specific task is called function defination.

Syntax:

```
<return type><function name>(<type of arguments>)
{
 <statement-1>
<statement-2>
return(<vlaxe>)
}
```

Example: program based upon function:

WAP to compute cube of a no. using function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int c,n;
int cube(int);
printf("Enter a no.");
scanf("%d",&n);
c=cube(n);
printf("cube of a no. is=%d",c);
}
int cube(int n)
{
c=n*n*n;
return(c);
}
```

WAP to compute factorial of a no. using function:

```
#include<stdio.h>
#include<conio.h>
void main()
```


Scanned with CamScanner

```
{
int n,f=1;
int fact(int)
printf("Enter a no.");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a no. is:=%d",f);
}
int fact(int n)
int f=1;
{
 for(int i=n;i>=n;i--)
{
 f=f*i;
}
return(f);
}
```

Recursion

Firstly, what is nested function?

When a function invokes another function then it is called nested function.

But,

When a function invokes itself then it is called recursion.

NOTE: In recursion, we must include a terminating condition so that it won't execute to infinite time.

Example: program based upon recursion:

WAP to compute factorial of a no. using Recursion:

```
#include<stdio.h>
#include<conio.h>
void main()
```


Scanned with CamScanner

```
{
int n,f;
int fact(int)
printf("Enter a no.");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a no. is:=%d",f);
}
int fact(int n)
int f=1;
{
 if(n=0)
return(f);
else
return(n*fact(n-1));
}
```

Passing parameters to a function:

Firstly, what are parameters?

parameters are the values that are passed to a function for processing.

There are 2 types of parameters:

a.) Actual Parameters.

b.) Formal Parameters.

a.) Actual Parameters:

These are the parameters which are used in main() function for function calling.

Syntax: <variable name>=<function name><actual argument>

Example: f=fact(n);

b.) Formal Parameters.

These are the parameters which are used in function defination for processing.

Scanned with CamScanner

Methods of parameters passing:

1.) Call by reference.

2.) Call by value.

1.) Call by reference:

In this method of parameter passing , original values of variables are passed from calling program to function.

Thus,

Any change made in the function can be reflected back to the calling program.

2.) Call by value.

In this method of parameter passing, duplicate values of parameters are passed from calling program to function defination.

Thus,

Any change made in function would not be reflected back to the calling program.

Example: Program based upon call by value:

```
# include<stdio.h>
# include<conio.h>
void main()
{
int a,b;
a=10;
b=20;
void swap(int,int)
printf("The value of a before swapping=%d",a);
printf("The value of b before swapping=%d",b);
void swap(a,b);
printf("The value of a after swapping=%d",a);
printf("The value of b after swapping=%d",b);
```


Scanned with CamScanner

```
}
void swap(int x, int y)
{
int t;
t=x;
x=y;
y=t;
}
```

STORAGE CLASSES

Every Variable in a program has memory associated with it.

Memory Requirement of Variables is different for different types of variables.

In C, Memory is allocated & released at different places

Term	Definition
Scope	Region or Part of Program in which Variable is accessible
Extent	Period of time during which memory is associated with variable
Storage Class	Manner in which memory is allocated by the Compiler for Variable **Different Storage Classes**

Storage class of variable Determines following things

Where the variable is stored

Scope of Variable

Default initial value

Lifetime of variable

A. Where the variable is stored:

Storage Class determines the location of variable, where it is declared. Variables declared with auto storage classes are declared inside main memory whereas variables declared with keyword register are stored inside the CPU Register.

Scanned with CamScanner

B. Scope of Variable

Scope of Variable tells compile about the visibility of Variable in the block. Variable may have Block Scope, Local Scope and External Scope. A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

C. Default Initial Value of the Variable

Whenever we declare a Variable in C, garbage value is assigned to the variable. Garbage Value may be considered as initial value of the variable. C Programming have different storage classes which has different initial values such as Global Variable have Initial Value as 0 while the Local auto variable have default initial garbage value.

D. Lifetime of variable

Lifetime of the = Time Of variable Declaration - Time of Variable Destruction

Suppose we have declared variable inside main function then variable will be destroyed only when the control comes out of the main .i.e end of the program.

Different Storage Classes:

Auto Storage Class

Static Storage Class

Extern Storage Class

Register Storage Class

Automatic (Auto) storage class

This is default storage class

All variables declared are of type Auto by default

In order to Explicit declaration of variable use 'auto' keyword

auto int num1 ; // Explicit Declaration

Features:

Storage	Memory
Scope	Local / Block Scope

Scanned with CamScanner

Life time	Exists as long as Control remains in the block
Default initial Value	Garbage

Example

```
void main()
{
 auto mum = 20 ;
 {
     auto num = 60 ;
          printf("nNum : %d",num);
 }
 printf("nNum : %d",num);
}
```

Output :

Num : 60

Num : 20

Note :

Two variables are declared in different blocks , so they are treated as different variables

External (extern) storage class in C Programming

Variables of this storage class are "Global variables"

Global Variables are declared outside the function and are accessible to all functions in the program

Generally , External variables are declared again in the function using keyword extern

In order to Explicit declaration of variable use 'extern' keyword

extern int num1 ;   // Explicit Declaration

Features :

Scanned with CamScanner

Storage	Memory
Scope	Global / File Scope
Life time	Exists as long as variable is running Retains value within the function
Default initial Value	Zero

Example

```
int num =  75 ;

void display();

void main()
{
 extern int num ;
          printf("nNum : %d",num);
 display();
}

void display()
{
 extern int num ;
          printf("nNum : %d",num);
}
```

Output :

Num : 75

Num : 75

Note :

Declaration within the function indicates that the function uses external variable

Functions belonging to same source code , does not require declaration (no need to write extern)

If variable is defined outside the source code , then declaration using extern keyword is required

Static Storage Class

Scanned with CamScanner

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

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

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```


Scanned with CamScanner

Register Storage Class

register keyword is used to define local variable.

Local variable are stored in register instead of **RAM**.

As variable is stored in register, the **Maximum size of variable = Maximum Size of Register**

unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.

This is generally used for **faster access**.

Common use is "**Counter**"

Syntax

```
{
register int count;
}
```

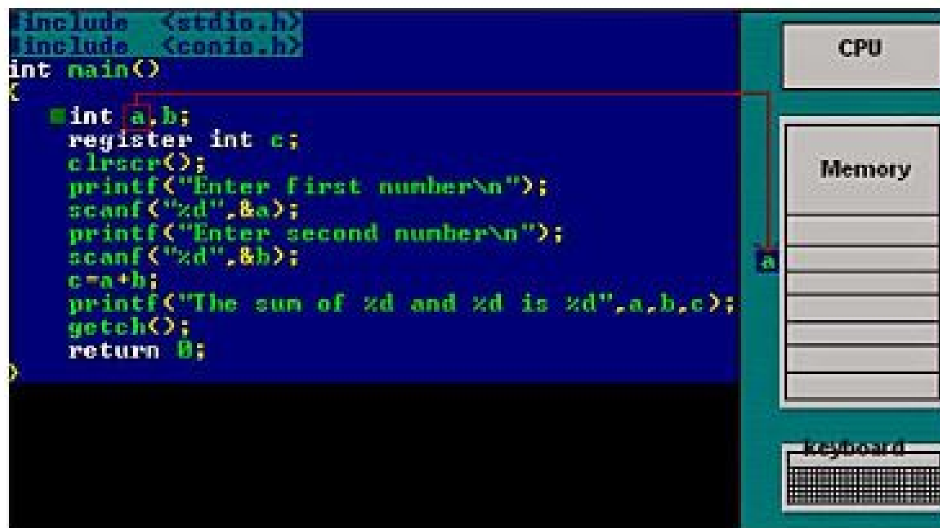
Register storage classes example

```
#include<stdio.h>

int main()
{
int num1,num2;
register int sum;

printf("\nEnter the Number 1 : ");
scanf("%d",&num1);

printf("\nEnter the Number 2 : ");
scanf("%d",&num2);

sum = num1 + num2;

printf("\nSum of Numbers : %d",sum);

return(0);
}
```

Explanation of program

Refer below animation which depicts the register storage classes –

Scanned with CamScanner



In the above program we have declared two variables num1,num2. These two variables are stored in RAM.

Another variable is declared which is stored in register variable.Register variables are stored in the register of the microprocessor.Thus memory access will be faster than other variables.

If we try to declare more register variables then it can treat variables asAuto storage variables as memory of microprocessor is fixed and limited.

Why we need Register Variable ?

Whenever we declare any variable inside C Program then memory will be randomly allocated at particular memory location.

We have to keep track of that memory location. We need to access value at that memory location using ampersand operator/Address Operator i.e (&).

If we store same variable in the register memory then we can access that memory location directly without using the Address operator.

Register variable will be accessed faster than the normal variable thus increasing the operation and program execution. Generally we use register variable as Counter.

Note : It is not applicable for arrays, structures or pointers.

Summary of register Storage class

Keyword	**register**
Storage Location	CPU Register

Scanned with CamScanner

Keyword	register
Initial Value	Garbage
Life	Local to the block in which variable is declared.
Scope	Local to the block.

Preprocessor directives

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Below is the list of preprocessor directives that C laxguage offers.

S.no	Preprocessor	Syntax	Description
1	Macro	#define	This macro defines constant value and can be any of the basic data types.
2	Header file inclusion	#include <file_name>	The source code of the file "file_name" is included in the main program at the specified place
3	Conditional compilation	#ifdef, #endif, #if, #else, #ifndef	Set of commands are included or excluded in source program before compilation with respect to the

Scanned with CamScanner

			condition
4	Other directives	#undef, #pragma	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.

EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C:

#define – This macro defines constant value and can be any of the basic data types.
#include <file_name> – The source code of the file "file_name" is included in the main C program where "#include <file_name>" is mentioned.

```
#include <stdio.h>

#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'

void main()
{
  printf("value of height    : %d \n", height );
  printf("value of number : %f \n", number );
  printf("value of letter : %c \n", letter );
  printf("value of letter_sequence : %s \n", letter_sequence);
  printf("value of backslash_char  : %c \n", backslash_char);

}
```

OUTPUT:

Scanned with CamScanner

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

"#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.

Otherwise, "else" clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
  #ifdef RAJU
  printf("RAJU is defined. So, this line will be added in " \
         "this C file\n");
  #else
  printf("RAJU is not defined\n");
  #endif
  return 0;
}
```

OUTPUT:

```
RAJU is defined. So, this line will be added in this C file
```

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

#ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file.

Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100
```


Scanned with CamScanner

```
int main()
{
  #ifndef SELVA
  {
    printf("SELVA is not defined. So, now we are going to " \
           "define here\n");
    #define SELVA 300
  }
  #else
  printf("SELVA is already defined in the program");

  #endif
  return 0;

}
```

OUTPUT:

```
SELVA is not defined. So, now we are going to define here
```

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

"If" clause statement is included in source file if given condition is true.

Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define a 100
int main()
{
  #if (a==100)
  printf("This line will be added in this C file since " \
         "a \= 100\n");
  #else
  printf("This line will be added in this C file since " \
         "a is not equal to 100\n");
  #endif
  return 0;
}
```

OUTPUT:

```
This line will be added in this C file since a = 100
```


Scanned with CamScanner

EXAMPLE PROGRAM FOR UNDEF IN C:

This directive undefines existing macro in the program.

```
#include <stdio.h>

#define height 100
void main()
{
  printf("First defined value for height    : %d\n",height);
  #undef height          // undefining variable
  #define height 600     // redefining the same for new value
  printf("value of height after undef \& redefine:%d",height);
}
```

OUTPUT:

```
First defined value for height : 100
value of height after undef & redefine : 600
```

EXAMPLE PROGRAM FOR PRAGMA IN C:

Pragma is used to call a function before and after main function in a C program.

```
#include <stdio.h>

void function1( );
void function2( );

#pragma startup function1
#pragma exit function2

int main( )
{
  printf ( "\n Now we are in main function" ) ;
  return 0;
}

void function1( )
{
  printf("\nFunction1 is called before main function call");
}

void function2( )
{
  printf ( "\nFunction2 is called just before end of " \
          "main fuxction" ) ;"
}
```


Scanned with CamScanner

OUTPUT:

```
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
```

MORE ON PRAGMA DIRECTIVE IN C:

S.no	Pragma command	description
1	#Pragma startup <function_name_1>	This directive executes function named "function_name_1" before
2	#Pragma exit <function_name_2>	This directive executes function named "function_name_2" just before termination of the program.
3	#pragma warn – rvl	If function doesn't return a value, then warnings are suppressed by this directive while compiling.
4	#pragma warn – par	If function doesn't use passed function parameter , then warnings are suppressed
5	#pragma warn – rch	If a non reachable code is written inside a program, such warnings are suppressed by this directive.

Scanned with CamScanner

POINTERS

Pointer Overview

Variable Name ➔	I	j	k
Value of Variable ➔	3	65524	65522
Address of Location ➔	65524	65522	65520

Consider above Diagram which clearly shows pointer concept in c programming –

i is the name given for particular memory location of ordinary variable.

Let us consider it's Corresponding address be 65624 and the Value stored in variable 'i' is 5

The address of the variable 'i' is stored in another integer variable whose name is 'j' and which is having corresponding address 65522

thus we can say that –

j = &i;

i.e

j = Address of i

Here j is not ordinary variable , It is special variable and called pointer variable as it stores the address of the another ordinary variable. We can summarize it like –

Variable Name	Variable Value	Variable Address
i	5	65524
j	65524	65522

B. C Pointer Basic Example:

```
#include <stdio.h>

int main()
{
 int *ptr, i;
 i = 11;

 /* address of i is assigned to ptr */
```


Scanned with CamScanner

```
  ptr = &i;

  /* show i's value using ptr variable */
  printf("Value of i : %d", *ptr);

  return 0;
}
```

See Output and Download »

You will get value of i = 11 in the above program.

C. Pointer Declaration Tips :

1. Pointer is declared with preceding * :

```
int *ptr;  //Here ptr is Integer Pointer Variable
int ptr;   //Here ptr is Normal  Integer Variable
```

2. Whitespace while Writing Pointer :

pointer variable name and asterisk can contain whitespace because whitespace is ignored by compiler.

```
int *ptr;
int       * ptr;
int *          ptr;
```

All the above syntax are legal and valid. We can insert any number of spaces or blanks inside declaration. We can also split the declaration on multiple lines.

D. Key points for Pointer :

Unline ordinary variables pointer is special type of variable which stores the address of ordinary variable.

Pointer can only store the whole or integer number because address of any type of variable is considered as integer.

It is good to initialize the pointer immediately after declaration

& symbol is used to get address of variable

* symbol is used to get value from the address given by pointer.

E. Pointer Summary :

Pointer is Special Variable used to Reference and de-reference memory. (*Will be covered in upcoming chapter)

Scanned with CamScanner

When we declare integer pointer then we can only store address of integer variable into that pointer.

Similarly if we declare character pointer then only the address of character variable is stored into the pointer variable.

Pointer storing the address of following DT	Pointer is called as
Integer	Integer Pointer
Character	Character Pointer
Double	Double Pointer
Float	Float Pointer

Pointer is a variable which stores the address of another variable

Since Pointer is also a kind of variable , thus pointer itself will be stored at different memory location.

2 Types of Variables :

Simple Variable that stores a value such as integer,float,character

Complex Variable that stores address of simple variable i.e pointer variables

Simple Pointer Example #1 :

```
#include<stdio.h>

int main()
{
int a = 3;
int *ptr;
ptr = &a;

return(0);
}
```

Explanation of Example :

Point	Variable 'a'	Variable 'ptr'
Name of Variable	a	ptr

Scanned with CamScanner

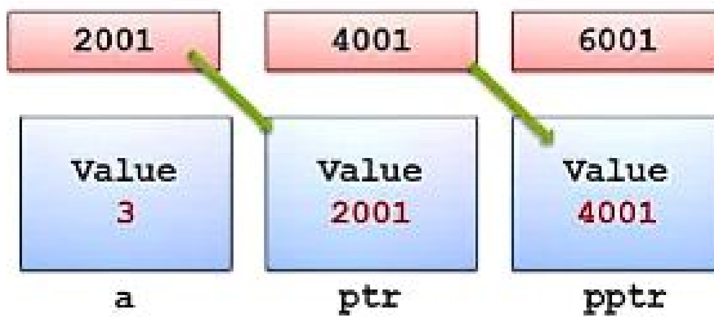
Point	Variable 'a'	Variable 'ptr'
Type of Value that it holds	Integer	Address of Integer 'a'
Value Stored	3	2001
Address of Variable	2001 (Assumption)	4001 (Assumption)

Simple Pointer Example #2 :

```
#include<stdio.h>

int main()
{
int a = 3;
int *ptr,**pptr;
ptr = &a;
pptr = &ptr;
return(0);
}
```

Explanation of Example

With reference to above program –



We have following associated points –

Point	Variable 'a'	Variable 'ptr'	Variable 'pptr'
Name of Variable	a	ptr	pptr
Type of Value that it holds	Integer	Address of 'a'	Address of 'ptr'
Value Stored	3	2001	4001

Scanned with CamScanner

Point	Variable 'a'	Variable 'ptr'	Variable 'pptr'
Address of Variable	2001	4001	6001

Pointer address operator in C Programming

Pointer address operator is denoted by '&' symbol

When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

lets take an example –

&n - It gives an address on variable n

Working of address operator

```
#include<stdio.h>
void main()
{
int n = 10;
printf("\nValue of n is : %d",n);
printf("\nValue of &n is : %u",&n);
}
```

Output :

Value of n is : 10

Value of &n is : 1002

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with %u

printf("\nValue of &n is : %u",&n);

Understanding address operator

Consider the following program –

```
#include<stdio.h>
int main()
{
int i = 5;
int *ptr;
```

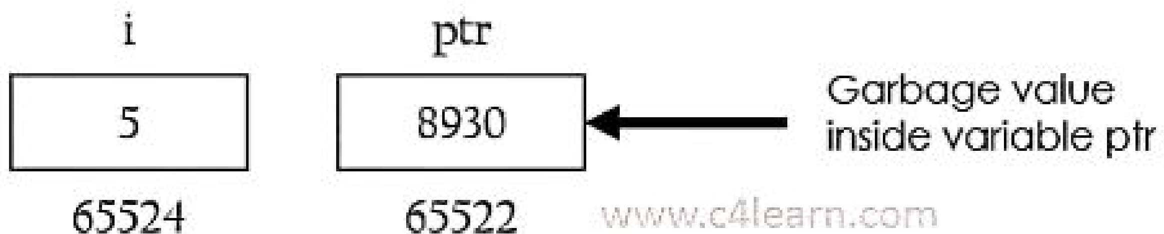

Scanned with CamScanner

```
ptr = &i;

printf("\nAddress of i    : %u",&i);
printf("\nValue of ptr is : %u",ptr);

return(0);
}
```

After declaration memory map will be like this –

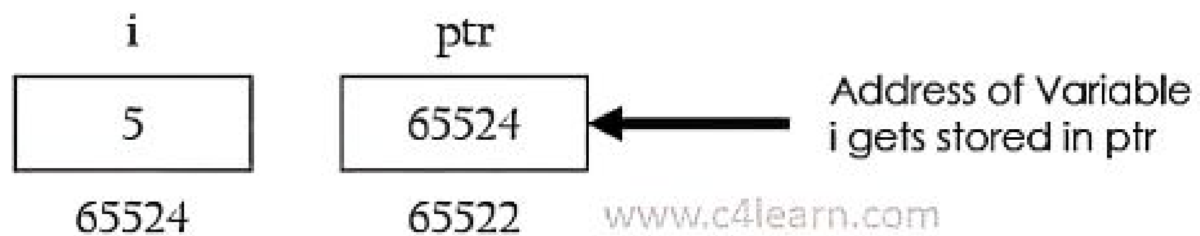
int i = 5;

int *ptr;



after Assigning the address of variable to pointer , i.e after the execution of this statement –

ptr = &i;



Invalid Use of pointer address operator

Address of literals

In C programming using address operator over literal will throw an error. We cannot use address operator on the literal to get the address of the literal.

&75

Only variables have an address associated with them, constant entity does not have corresponding address. Similarly we cannot use address operator over character literal –

&('a')

Character 'a' is literal, so we cannot use address operator.

Address of expressions

Scanned with CamScanner

(a+b) will evaluate addition of values present in variables and output of (a+b)is nothing but Literal, so we cannot use Address operator

&(a+b)

Memory Organization for Pointer Variable:

When we use variable in program then Compiler keeps some memory for that variable depending on the **data type**

The address given to the variable is Unique with that variable name

When Program execution starts the **variable name** is automatically translated into the corresponding **address**.

Explanation :

Pointer Variable is nothing but a memory address which holds another address .

In the above program "i" is name given for memory location for human understanding , but compiler is unable to recognize "i" . Compiler knows only address.

In the next chapter we will be learning , Memory requirement for storing pointer variable.

Syntax for Pointer Declaration in C :

data_type *<pointer_name>;

Explanation :

data_type

Type of variable that the pointer **points to**

OR data type whose address is stored in **pointer_name**

Asterisk(*)

Asterisk is called as **Indirection Operator**

It is also called as **Value at address Operator**

It Indicates **Variable declared is of Pointer type**

Scanned with CamScanner

pointer_name

Must be any **Valid C identifier**

Must follow all Rules of Variable name declaration

Ways of Declaring Pointer Variable:

[box] * can appears anywhere between Pointer_name and Data Type

```
int *p;
int *       p;
int    *  p;
```

Example of Declaring Integer Pointer:

```
int n = 20;
int *ptr;
```

Example of Declaring Character Pointer:

```
char ch = 'A';
char *cptr;
```

Example of Declaring Float Pointer:

```
float fvar = 3.14;
float *fptr;
```

How to Initialize Pointer in C Programming?

```
pointer = &variable;
```

Above is the syntax for initializing pointer variable in C.

Initialization of Pointer can be done using following 4 Steps :

Declare a Pointer Variable and Note down the Data Type.

Declare another Variable with Same Data Type as that of Pointer Variable.

Initialize Ordinary Variable and assign some value to it.

Now Initialize pointer by assigning the address of ordinary variable to pointer variable.

below example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
{

int a;       // Step 1
int *ptr;    // Step 2
```


Scanned with CamScanner

```
a = 10;      // Step 3
ptr = &a;    // Step 4

return(0);
}
```

Explanation of Above Program :

Pointer should not be used before initialization.

"ptr" is pointer variable used to store the address of the variable.

Stores address of the variable **'a'** .

Now "ptr" will contain the address of the variable "a" .

Note :

[box]Pointers are always initialized before using it in the program[/box]

Example : Initializing Integer Pointer

```
#include<stdio.h>
int main()
{
int a = 10;
int *ptr;

ptr = &a;
printf("\nValue of ptr : %u",ptr);

return(0);
}
```

Output :

Value of ptr : 4001

Pointer arithematic

Incrementing Pointer:

Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.

Incrementing Pointer Variable Depends Upon data type of the Pointer variable

Formula : (After incrementing)

new value = current address + i * size_of(data type)

Three Rules should be used to increment pointer –

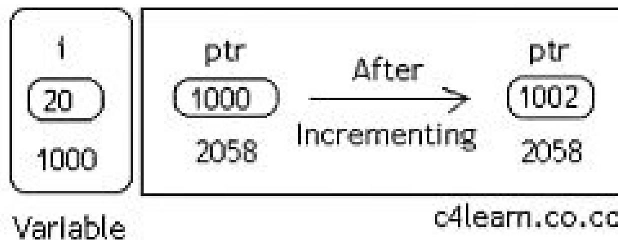
Scanned with CamScanner

Address + 1 = Address

Address++ = Address

++Address = Address

Pictorial Representation :



Data Type	Older Address stored in pointer	Next Address stored in pointer after incrementing (ptr++)
int	1000	1002
float	1000	1004
char	1000	1001

Explanation : Incremeting Pointer

Incrementing a pointer to an integer data will cause its **value to be incremented by 2** .

This differs from compiler to compiler as memory required to store integer **vary compiler to compiler**

[box]**Note to Remember :** Increment and Decrement Operations on pointer should be used when we have Continues memory (in Array).[/box]

Live Example 1 : Increment Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000;

ptr=ptr+1;
printf("New Value of ptr : %u",ptr);
```


Scanned with CamScanner

```
return 0;
}
```

Output :

New Value of ptr : 1002

Live Example 2 : Increment Double Pointer

```
#include<stdio.h>

int main(){

double *ptr=(double *)1000;

ptr=ptr+1;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

Output :

New Value of ptr : 1004

Live Example 3 : Array of Pointer

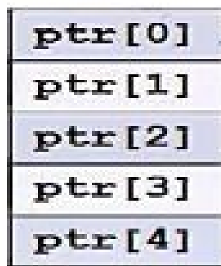
```
#include<stdio.h>

int main(){

float var[5]={1.1f,2.2f,3.3f};
float(*ptr)[5];

ptr=&var;
printf("Value inside ptr : %u",ptr);

ptr=ptr+1;
printf("Value inside ptr : %u",ptr);

return 0;
}
```

Output :

Value inside ptr : 1000

Value inside ptr : 1020

Scanned with CamScanner



Explanation :

Address of ptr[0] = 1000

We are storing Address of float array to ptr[0]. –

Address of ptr[1]

= Address of ptr[0] + (Size of Data Type)*(Size of Array)

= 1000 + (4 bytes) * (5)

= 1020

Address of Var[0]…Var[4] :

Address of var[0] = 1000

Address of var[1] = 1004

Address of var[2] = 1008

Address of var[3] = 1012

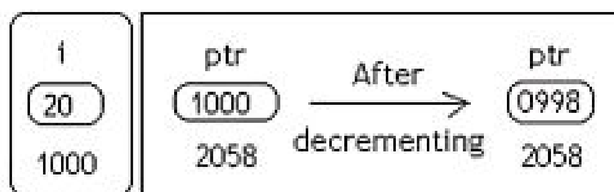
Address of var[4] = 1016

Formula : (After decrementing)

new_address = (current address) - i * size_of(data type)

[box]Decrementation of Pointer Variable Depends Upon : data type of the Pointer variable[/box]

Example :



Scanned with CamScanner

Data Type	Older Address stored in pointer	Next Address stored in pointer after incrementing (ptr--)
int	1000	0998
float	1000	0996
char	1000	0999

Explanation:

Decrementing a pointer to an integer data will cause its value to be decremented by 2

This differs from compiler to compiler as memory required to store integer vary **compiler to compiler**

Pointer Program: Difference between two integer Pointers

```
#include<stdio.h>

int main(){

float *ptr1=(float *)1000;
float *ptr2=(float *)2000;

printf("\nDifference : %d",ptr2-ptr1);

return 0;
}
```

Output :

Difference : 250

Explanation :

Ptr1 and Ptr2 are two pointers which holds memory address of Float Variable.

Ptr2-Ptr1 will gives us number of floating point numbers that can be stored.

ptr2 - ptr1 = (2000 - 1000) / sizeof(float)

= 1000 / 4

= 250

Live Example 2:

```
#include<stdio.h>

struct var{
```


Scanned with CamScanner

```
    char cvar;
    int ivar;
    float fvar;
};

int main(){

struct var *ptr1,*ptr2;

ptr1 = (struct var *)1000;
ptr2 = (struct var *)2000;

printf("Difference= %d",ptr2-ptr1);

return 0;
}
```

Output :

Difference = 142

Explanation :

ptr2-ptr1 = (2000 - 1000)/Sizeof(struct var)

= 1000 / (1+2+4)

= 1000 / 7

= 142

Adding integer value with Pointer

In C Programming we can add any integer number to Pointer variable. It is perfectly legal in c programming to add integer to pointer variable.

In order to compute the final value we need to use following formulae :

final value = (address) + (number * size of data type)

Consider the following example –

```
int *ptr , n;

ptr = &n ;

ptr = ptr + 3;
```

Live Example 1 : Increment Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000;
```


Scanned with CamScanner

```
ptr=ptr+3;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

Output :

New Value of ptr : 1006

Explanation of Program :

In the above program –

int *ptr=(int *)1000;

this line will store 1000 in the pointer variable considering 1000 is memory location for any of the integer variable.

Formula :

ptr = ptr + 3 * (sizeof(integer))
   = 1000 + 3 * (2)
   = 1000 + 6
   = 1006

Similarly if we have written above statement like this –

float *ptr=(float *)1000;

then result may be

ptr = ptr + 3 * (sizeof(float))
   = 1000 + 3 * (4)
   = 1000 + 12
   = 1012

Suppose we have subtracted "n" from pointer of any data type having initial addess as "init_address" then after subtraction we can write –

ptr = initial_address - n * (sizeof(data_type))

Subtracting integer value with Pointer

int *ptr , n;

ptr = &n ;

ptr = ptr - 3;

Scanned with CamScanner

Live Example 1 : Decrement Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000;

ptr=ptr-3;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

Output :

New Value of ptr : 994

Formula :

```
ptr = ptr   - 3 * (sizeof(integer))
    = 1000  - 3 * (2)
    = 1000  - 6
    = 994
```

Summary :

Pointer - Pointer = Integer

Pointer - Integer = Pointer

Differencing Pointer in C Programming Language :

Differencing Means **Subtracting two Pointers**.

Subtraction gives the Total number of objects between them .

Subtraction indicates "How apart the two Pointers are ?"

C Program to Compute Difference Between Pointers :

```
#include<stdio.h>

int main()
{
```


Scanned with CamScanner

```
int num , *ptr1 ,*ptr2 ;

ptr1 = &num ;
ptr2 = ptr1 + 2 ;

printf("%d",ptr2 - ptr1);

return(0);
}
```

Output :

2

ptr1 stores the **address of Variable num**

Value of ptr2 is incremented by **4 bytes**

Differencing two Pointers

Important Observations :

Suppose the Address of Variable num = 1000.

Statement	Value of Ptr1	Value of Ptr2
int num , *ptr1 ,*ptr2 ;	Garbage	Garbage
ptr1 = &num ;	1000	Garbage
ptr2 = ptr1 + 2 ;	1000	1004
ptr2 - ptr1	1000	1004

Computation of Ptr2 – Ptr1 :

Remember the following formula while computing the difference between two pointers –

Final Result = (ptr2 - ptr1) / Size of Data Type

Step 1 : Compute Mathematical Difference (Numerical Difference)

ptr2 - ptr1 = Value of Ptr2 - Value of Ptr1

= 1004 - 1000

= 4

Step 2 : Finding Actual Difference (Technical Difference)

Final Result = 4 / Size of Integer

= 4 / 2

= 2

Scanned with CamScanner

Numerically Subtraction (ptr2-ptr1) differs by 4

As both are Integers they are numerically Differed by 4 and Technically by 2 objects

Suppose Both pointers of float the they will be differed numerically by 8 and Technically by 2 objects

Consider the below statement and refer the following table –

int num = ptr2 - ptr1;

and

If Two Pointers are of Following Data Type	Numerical Difference	Technical Difference
Integer	2	1
Float	4	1
Character	1	1

Comparison between two Pointers :

Pointer comparison is Valid only if the **two pointers are Pointing to same array**

All Relational Operators can be used for comparing pointers of **same type**

All Equality and Inequality Operators can be used with all Pointer types

Pointers **cannot be Divided or Multiplied**

Point 1 : Pointer Comparison

```
#include<stdio.h>

int main()
{
int *ptr1,*ptr2;

ptr1 = (int *)1000;
ptr2 = (int *)2000;

if(ptr2 > ptr1)
   printf("Ptr2 is far from ptr1");

return(0);
}
```


Scanned with CamScanner

Pointer Comparison of Different Data Types :

```
#include<stdio.h>

int main()
{
int *ptr1;
float *ptr2;

ptr1 = (int *)1000;
ptr2 = (float *)2000;

if(ptr2 > ptr1)
   printf("Ptr2 is far from ptr1");

return(0);
}
```

Explanation :

Two Pointers of different data types can be compared .

In the above program we have compared two pointers of different data types.

It is perfectly **legal in C Programming.**

[box]As we know Pointers can store Address of any data type, address of the data type is "Integer" so we can compare address of any two pointers although they are of different data types.[/box]

Following operations on pointers :

>	Greater Than
<	Less Than
>=	Greater Than And Equal To
<=	Less Than And Equal To
==	Equals
!=	Not Equal

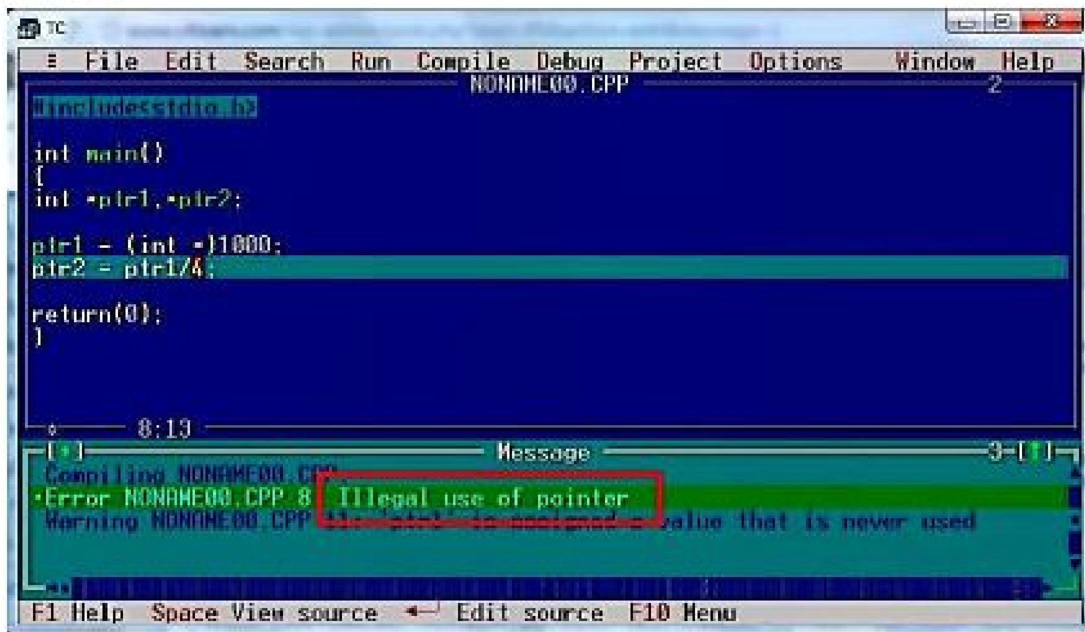
Divide and Multiply Operations :

```
#include<stdio.h>
```


Scanned with CamScanner

```
int main()
{
int *ptr1,*ptr2;

ptr1 = (int *)1000;
ptr2 = ptr1/4;

return(0);
}
```

Output :



Pointer to pointer

Pointer to Pointer in C Programming

Declaration : Double Pointer

```
int  **ptr2ptr;
```

Consider the Following Example :

```
int num = 45 , *ptr , **ptr2ptr ;
ptr = &num;
ptr2ptr = &ptr;
```

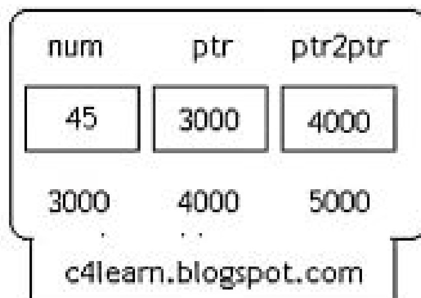
What is Pointer to Pointer ?

Scanned with CamScanner

Double (**) is used to denote the **double Pointer**

Pointer Stores the address of the Variable

Double Pointer **Stores the address of the Pointer Variable**



Statement	What will be the Output ?
*ptr	45
**ptr2ptr	45
ptr	&n
ptr2ptr	&ptr

Notes :

Conceptually we can have Triple ….. n pointers

Example : *****n,****b can be another example

Live Example :

```
#include<stdio.h>

int main()
{
int num = 45 , *ptr , **ptr2ptr ;
ptr     = &num;
ptr2ptr = &ptr;

printf("%d",**ptr2ptr);
```


Scanned with CamScanner

```
return(0);
}
```

Output :

45

Scanned with CamScanner