

Pointers & Address



- Explain what a pointer is and where it is used
- Assign values to pointers
- Pointer variables and pointer operators
- Pointer Arithmetic & Comparisons
- Pointers and Arrays

Section 1

WHAT IS A POINTER?



Pointers



What is a Pointer?

- A pointer is a variable, which contains the address of a memory location of another variable
- If one variable contains the address of another variable, the first variable is said to point to the second variable
- A pointer provides an indirect method of accessing the value of a data item
- Pointers can point to variables of other fundamental data types like int, char, or double or data aggregates like arrays or structures

What are Pointers used for?

Some situations where pointers can be used are:

- To return more than one value from a function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them instead of moving the arrays itself
- To allocate memory and access it (Direct Memory Allocation)

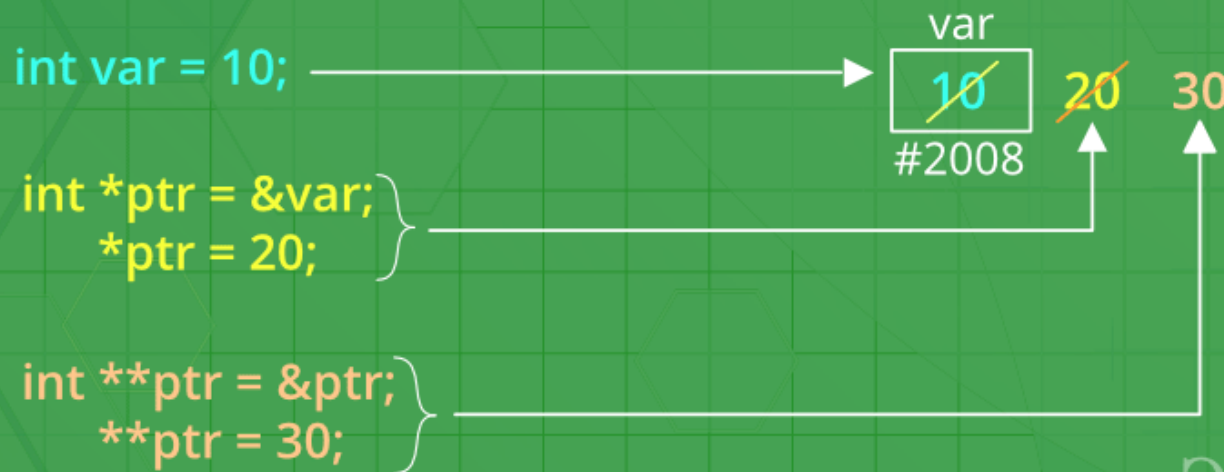
- Here is how we can declare pointers:

```
int* p;
```

- We have declared a pointer p of int type. You can also declare pointers in these ways:

```
int *p1;  
int * p2;
```

How pointer works in C



Section 2

ASSIGN VALUES TO POINTERS

Assigning Values To Pointers - 1

- Values can be assigned to pointers through the **&** operator.

```
ptr_var = &var;
```

- Here the address of var is stored in the variable ptr_var
- It is also possible to assign values to pointers through another pointer variable pointing to a data item of the same data type

```
ptr_var = &var;
```

```
ptr_var2 = ptr_var;
```

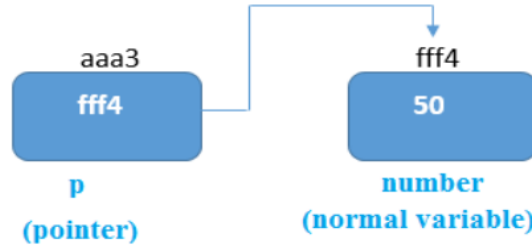
- Variables can be assigned values through their pointers as well

***ptr_var = 10;**

- The above declaration will assign 10 to the variable var if ptr_var points to var

Pointer example

- An example of using pointers to print the address and value is given below:




- In the above figure, pointer variable stores the address of number variable, i.e., **fff4**. The value of number variable is **50**. But the address of pointer variable **p** is **aaa3**.
- By the help of * (**indirection operator**), we can print the value of pointer variable **p**.

Pointer example - 1

- An example of using pointers to print the address and value is given below:

```
#include<stdio.h>
int main()
{
    int number=50;
    int *p;
    // stores the address of number variable
    p = &number;
    // p contains the address of the number therefore printing p gives the address of number.
    printf("Address of p variable is %x \n", p);
    // * is used to dereference a pointer therefore if print *p,
    // the value stored at the address contained by p.
    printf("Value of p variable is %d \n", *p);
    return 0;
}
```



Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50

Section 3

POINTER VARIABLES AND POINTER OPERATORS

- A pointer declaration consists of a base type and a variable name preceded by an *

✓ **General declaration syntax is :**

type *name;

✓ **For Example:**

int *var2;

- There are 2 special operators which are used with pointers:

& and *****

- The **&** operator is a unary operator and it returns the memory address of the operand

var2 = &var1;

- The second operator ***** is the complement of **&**. It is a unary operator and returns the value contained in the memory location pointed to by the pointer variable's value

temp = *var2;

Pointer variables and pointer operators

/* 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;
}
```

- To access address of a variable to a pointer, use the unary operator **&** (**ampersand**) that returns the address of that variable. For example **&x** gives address of variable **x**.

Pointer variables and pointer operators - 1

One more operator is **unary** * (Asterisk) which is used for two things: To declare a pointer variable. When a pointer variable is declared in C/C++, there must be a * before its name:

// C program to demonstrate declaration of pointer variables.

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

```
int main() {
```

```
    int x = 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 = &x;
```

```
    return 0;
```

```
}
```

Pointer variables and pointer operators - 2

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**.

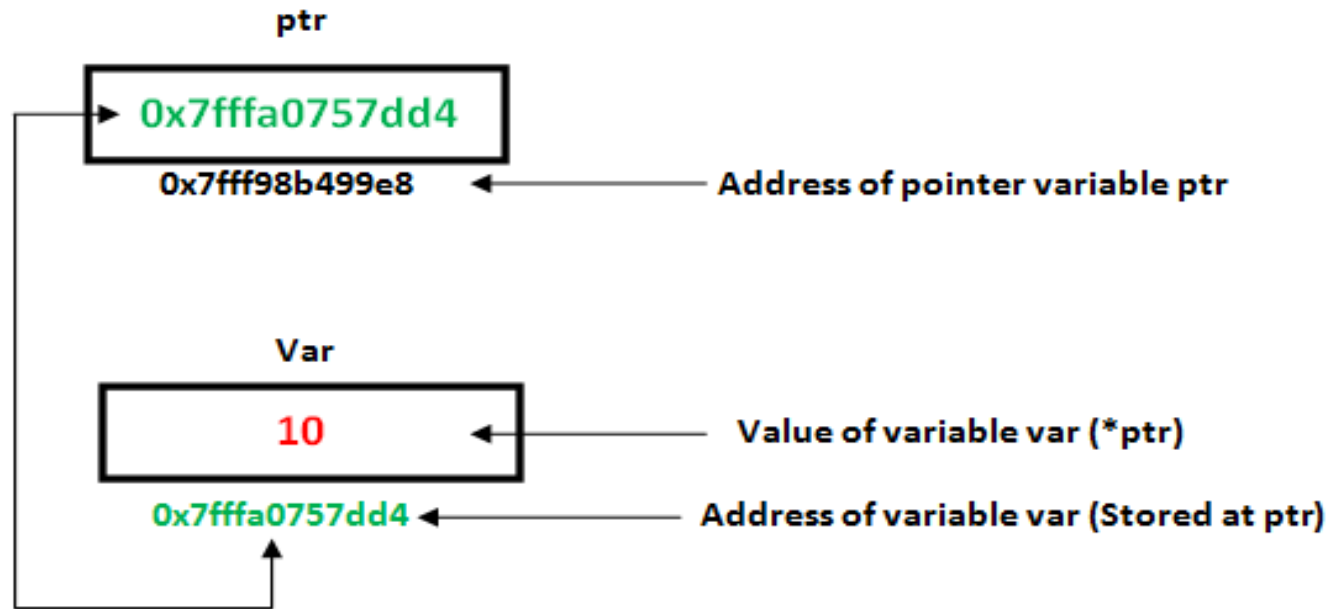
```
// C program to demonstrate use of * for pointers in C
#include <stdio.h>
int main() {
    int Var = 10; // A normal integer variable
    int *ptr = &Var; // A pointer variable that holds address of var.
    // This line prints value at address stored in ptr. Value stored is value of variable "var"
    printf("Value of Var = %d\n", *ptr);
    // The output of this line may be different in different runs even on same machine.
    printf("Address of Var = %p\n", ptr);
    // Use ptr as a left hand side of assignment
    *ptr = 20; // Value at address is now 20
    printf("After doing *ptr = 20, *ptr is %d\n", *ptr);
    return 0;
}
```



Value of Var = 10
Address of Var = 0x7ffa057dd4
After doing *ptr = 20, *ptr is 20

Pointer variables and pointer operators - 3

Below is pictorial representation of above program:



Section 4

POINTER ARITHMETIC & COMPARISONS

- Addition and subtraction are the only operations that can be performed on pointers

```
int var, *ptr_var;  
ptr_var = &var;  
var = 500;  
ptr_var++ ;
```

- Let us assume that **var** is stored at the address **1000**
- Then **ptr_var** has the value **1000** stored in it. Since integers are **2** bytes long, after the expression “**ptr_var++;**” **ptr_var** will have the value as **1002** and not **1001**

Pointer Arithmetic - 2

<code>++ptr_var</code> or <code>ptr_var++</code>	points to next integer after var
<code>--ptr_var</code> or <code>ptr_var--</code>	points to integer previous to var
<code>ptr_var + i</code>	points to the <i>i</i> th integer after var
<code>ptr_var - i</code>	points to the <i>i</i> th integer before var
<code>++*ptr_var</code> or <code>(*ptr_var)++</code>	will increment var by 1
<code>*ptr_var++</code>	will fetch the value of the next integer after var

- Each time a pointer is incremented, it points to the memory location of the next element of its base type
- Each time it is decremented it points to the location of the previous element
- All other pointers will increase or decrease depending on the length of the data type they are pointing to

Example about increment arithmetic:



Pointer Arithmetic - 4

```
// C++ program to illustrate Pointer Arithmetic
// in C/C++
#include <bits/stdc++.h>

// Driver program
int main()
{
    // Declare an array
    int v[3] = {10, 100, 200};

    // Declare pointer variable
    int *ptr;

    // Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++)
    {
        printf("Value of *ptr = %d\n", *ptr);
        printf("Value of ptr = %p\n\n", ptr);

        // Increment pointer ptr by 1
        ptr++;
    }
}
```



```
Output:Value of *ptr = 10
Value of ptr = 0x7ffcae30c710

Value of *ptr = 100
Value of ptr = 0x7ffcae30c714

Value of *ptr = 200
Value of ptr = 0x7ffcae30c718
```

Pointer Comparisons

- Two pointers can be compared in a relational expression provided both the pointers are pointing to variables of the same type
- Consider that `ptr_a` and `ptr_b` are 2 pointer variables, which point to data elements `a` and `b`. In this case the following comparisons are possible:

<code>ptr_a < ptr_b</code>	Returns true provided a is stored before b
<code>ptr_a > ptr_b</code>	Returns true provided a is stored after b
<code>ptr_a <= ptr_b</code>	Returns true provided a is stored before b or <code>ptr_a</code> and <code>ptr_b</code> point to the same location
<code>ptr_a >= ptr_b</code>	Returns true provided a is stored after b or <code>ptr_a</code> and <code>ptr_b</code> point to the same location.
<code>ptr_a == ptr_b</code>	Returns true provided both pointers <code>ptr_a</code> and <code>ptr_b</code> points to the same data element.
<code>ptr_a != ptr_b</code>	Returns true provided both pointers <code>ptr_a</code> and <code>ptr_b</code> point to different data elements but of the same type.
<code>ptr_a == NULL</code>	Returns true if <code>ptr_a</code> is assigned NULL value (zero)

Section 5

POINTERS AND ARRAYS

- The address of an array element can be expressed in two ways :
 - ✓ By writing the actual array element preceded by the ampersand sign (&)
 - ✓ By writing an expression in which the subscript is added to the array name

Pointers and Single Dimensional Arrays-2

// C program to illustrate Array Name as Pointers in C

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

```
int main()
```

```
{
```

```
    int val[3] = { 5, 10, 15}; // Declare an array
```

```
    int *ptr; // Declare pointer variable
```

```
    // Assign address of val[0] to ptr
```

```
    ptr = val ; // or use: ptr=&val[0]; (both are same)
```

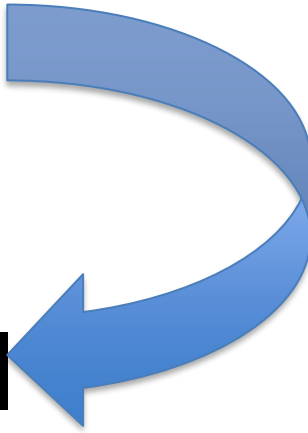
```
    printf( "Elements of the array are: " );
```

```
    printf( "%d %d %d", ptr[0], ptr[1], ptr[2] );
```

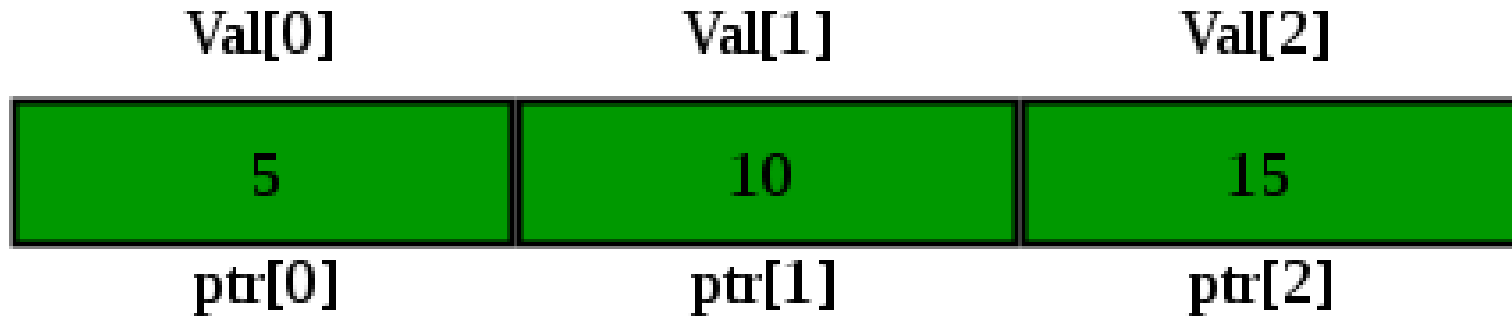
```
    return 0;
```

```
}
```

Elements of the array are: 5 10 15



The positions of the elements in the above array in virtual memory are shown with the following figure:



Pointers and Single Dimensional Arrays-4

```
#include<stdio.h>

void main()
{
    int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for (int i = 0; i < 10; i ++)
    {
        // print value of each element in the array
        printf("\ni=%d, ary[i]=%d, *(ary+i)=%d", &i, ary[i], *(ary + i));
        printf("&ary[i]= %X, ary+i=%X", &ary[i], ary+i);
        // %X gives unsigned hexadecimal
    }
}
```

Output:

i=0	ary[i]=1	*(ary+i)=1	&ary[i]=194	ary+i = 194
i=1	ary[i]=2	*(ary+i)=2	&ary[i]=196	ary+i = 196
i=2	ary[i]=3	*(ary+i)=3	&ary[i]=198	ary+i = 198
i=3	ary[i]=4	*(ary+i)=4	&ary[i]=19A	ary+i = 19A
i=4	ary[i]=5	*(ary+i)=5	&ary[i]=19C	ary+i = 19C
i=5	ary[i]=6	*(ary+i)=6	&ary[i]=19E	ary+i = 19E
i=6	ary[i]=7	*(ary+i)=7	&ary[i]=1A0	ary+i = 1A0
i=7	ary[i]=8	*(ary+i)=8	&ary[i]=1A2	ary+i = 1A2
i=8	ary[i]=9	*(ary+i)=9	&ary[i]=1A4	ary+i = 1A4
i=9	ary[i]=10	*(ary+i)=10	&ary[i]=1A6	ary+i = 1A6

- A two-dimensional array can be defined as a pointer to a group of contiguous one-dimensional arrays
- A two-dimensional array declaration can be written as :

`data_type (*ptr_var) [expr 2];`

instead of

`data_type ptr_var [expr1] [expr 2];`

- Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration. Look at example below:

```
int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };
```

- In general, **nums[i][j]** is equivalent to ***(* (nums+i)+j)**

<u>Pointer Notation</u>	<u>Array Notation</u>	<u>Value</u>
<code>*(*nums)</code>	<code>nums[0][0]</code>	16
<code>*(*nums + 1)</code>	<code>nums[0][1]</code>	18
<code>*(*nums + 2)</code>	<code>nums[0][2]</code>	20
<code>*(* (nums + 1))</code>	<code>nums[1][0]</code>	25
<code>*(* (nums + 1) + 1)</code>	<code>nums[1][1]</code>	26
<code>*(* (nums + 1) + 2)</code>	<code>nums[1][2]</code>	27

Thank you

Q&A

