Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So, it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>

int main () {

  int  var1;
  char var2[10];

  printf("Address of var1 variable: %x\n", &var1  );
  printf("Address of var2 variable: %x\n", &var2  );

  return 0;
}
```

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

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer

and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include <stdio.h>

int main () {

  int  var = 20;   /* actual variable declaration */
  int  *ip;        /* pointer variable declaration */

  ip = &var;  /* store address of var in pointer variable*/

  printf("Address of var variable: %x\n", &var  );

  /* address stored in pointer variable */
  printf("Address stored in ip variable: %x\n", ip );

  /* access the value using the pointer */
  printf("Value of *ip variable: %d\n", *ip );

  return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Pointer Syntax

Here is how we can declare pointers.

```
1.  int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
1.  int *p1;
2.  int * p2;
```

Let's take another example of declaring pointers.

```
1.  int* p1, p2;
```

Here, we have declared a pointer `p1` and a normal variable `p2`.

Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;
c = 5;
pc = &c;
```

Here, 5 is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);  // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

Note: In the above example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c;`

By the way, `*` is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

Changing Value Pointed by Pointers

Let's take an example.

```
int* pc, c;
c = 5;
pc = &c;
c = 1;
printf("%d", c);   // Output: 1
printf("%d", *pc);  // Ouptut: 1
```

We have assigned the address of `c` to the `pc` pointer.
Then, we changed the value of `c` to 1. Since `pc` and the address of `c` is the same, `*pc` gives us 1.

Let's take another example.

```
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);  // Ouptut: 1
printf("%d", c);   // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.
Then, we changed `*pc` to 1 using `*pc = 1;`. Since `pc` and the address of `c` is the same, `c` will be equal to 1.

Let's take one more example.

```
int* pc, c, d;
```

```
c = 5;
d = -15;

pc = &c; printf("%d", *pc); // Output: 5
pc = &d; printf("%d", *pc); // Ouptut: -15
```

Initially, the address of `c` is assigned to the `pc` pointer using `pc = &c;`. Since `c` is 5, `*pc` gives us 5.
Then, the address of `d` is assigned to the `pc` pointer using `pc = &d;`. Since `d` is -15, `*pc` gives us -15.

Example: Working of Pointers

Let's take a working example.

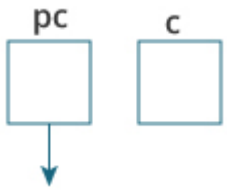
```
#include <stdio.h>
int main()
{
  int* pc, c;

  c = 22;
  printf("Address of c: %p\n", &c);
  printf("Value of c: %d\n\n", c);  // 22

  pc = &c;
  printf("Address of pointer pc: %p\n", pc);
  printf("Content of pointer pc: %d\n\n", *pc); // 22

  c = 11;
  printf("Address of pointer pc: %p\n", pc);
  printf("Content of pointer pc: %d\n\n", *pc); // 11

  *pc = 2;
  printf("Address of c: %p\n", &c);
  printf("Value of c: %d\n\n", c); // 2
  return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

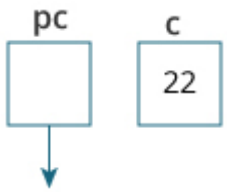
Explanation of the program

1. `int* pc, c;`



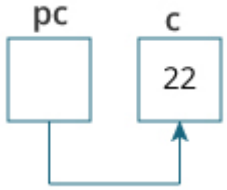
Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created.
Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`



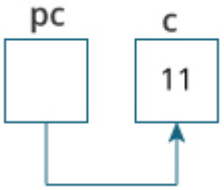
This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`



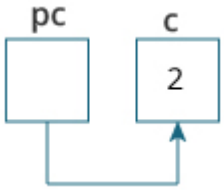
This assigns the address of variable `c` to the pointer `pc`.

4. `c = 11;`



This assigns 11 to variable `c`.

5. `*pc = 2;`



This change the value at the memory location pointed by the pointer `pc` to 2.

Common mistakes when working with pointers

Suppose, you want pointer `pc` to point to the address of `c`. Then,

```
int c, *pc;

// pc is address but c is not
pc = c; // Error

// &c is address but *pc is not
*pc = &c; // Error

```

```
// both &c and pc are addresses
pc = &c;

// both c and *pc values
*pc = c;
```

Here's an example of pointer syntax beginners often find confusing.

```
#include <stdio.h>
int main() {
  int c = 5;
  int *p = &c;

  printf("%d", *p);  // 5
  return 0;
}
```

Why didn't we get an error when using `int *p = &c;`**?**

It's because

```
int *p = &c;
```

is equivalent to

```
int *p:
p = &c;
```

In both cases, we are creating a pointer `p` (not `*p`) and assigning `&c` to it.

To avoid this confusion, we can use the statement like this:

```
int* p = &c;
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>

int main () {

  int  *ptr = NULL;

  printf("The value of ptr is : %x\n", ptr  );

  return 0;
}
```

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

The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

C - Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array –

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = var;

  for ( i = 0; i < MAX; i++) {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the next location */
    ptr++;
  }
```

```
  return 0;
}
```

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

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below –

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = &var[MAX-1];

  for ( i = MAX; i > 0; i--) {

    printf("Address of var[%d] = %x\n", i-1, ptr );
    printf("Value of var[%d] = %d\n", i-1, *ptr );

    /* move to the previous location */
    ptr--;
  }

  return 0;
}
```

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

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

```
#include <stdio.h>

const int MAX = 3;

int main () {

  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have address of the first element in pointer */
  ptr = var;
  i = 0;

  while ( ptr <= &var[MAX - 1] ) {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* point to the next location */
    ptr++;
    i++;
  }

  return 0;
}
```

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

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Relationship Between Arrays and Pointers

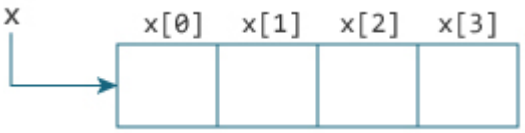
An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
1.  #include <stdio.h>
2.  int main() {
3.    int x[4];
4.    int i;
5.
6.    for(i = 0; i < 4; ++i) {
7.      printf("&x[%d] = %p\n", i, &x[i]);
8.    }
9.
10.   printf("Address of array x: %p", x);
11.
12.   return 0;
13. }
```

Output

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).
Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.



From the above example, it is clear that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example 1: Pointers and Arrays

```
1.  #include <stdio.h>
2.  int main() {
3.   int i, x[6], sum = 0;
4.   printf("Enter 6 numbers: ");
5.   for(i = 0; i < 6; ++i) {
6.   // Equivalent to scanf("%d", &x[i]);
7.      scanf("%d", x+i);
8.
9.   // Equivalent to sum += x[i]
10.     sum += *(x+i);
11.  }
12.  printf("Sum = %d", sum);
13.  return 0;
14. }
```

When you run the program, the output will be:

```
Enter 6 numbers:  2
 3
 4
 4
 12
 4
Sum = 29
```

Here, we have declared an array `x` of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that **pointers and arrays are not the same**.
There are a few cases where array names don't decay to pointers. To learn more, visit: [When does array name doesn't decay into a pointer?]

Example 2: Arrays and Pointers

```
#include <stdio.h>
int main() {
  int x[5] = {1, 2, 3, 4, 5};
  int* ptr;

  // ptr is assigned the address of the third element
  ptr = &x[2];

  printf("*ptr = %d \n", *ptr);   // 3
  printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
  printf("*(ptr-1) = %d", *(ptr-1));  // 2

  return 0;
}
```

When you run the program, the output will be:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

In this example, `&x[2]`, the address of the third element, is assigned to the `ptr` pointer. Hence, `3` was displayed when we printed `*ptr`.
And, printing `*(ptr+1)` gives us the fourth element. Similarly, printing `*(ptr-1)` gives us the second element.

Passing pointers to functions in C

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function –

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

void getSeconds(unsigned long *par);

int main () {

  unsigned long sec;
```

```
  getSeconds( &sec );

  /* print the actual value */
  printf("Number of seconds: %ld\n", sec );

  return 0;
}

void getSeconds(unsigned long *par) {
  /* get the current number of seconds */
  *par = time( NULL );
  return;
}
```

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

```
Number of seconds :1294450468
```

The function, which can accept a pointer, can also accept an array as shown in the following example –

```
#include <stdio.h>

/* function declaration */
double getAverage(int *arr, int size);

int main () {

  /* an int array with 5 elements */
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg;

  /* pass pointer to the array as an argument */
  avg = getAverage( balance, 5 ) ;

  /* output the returned value  */
  printf("Average value is: %f\n", avg );
  return 0;
}

double getAverage(int *arr, int size) {

  int  i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }

  avg = (double)sum / size;
  return avg;
}
```

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

```
Average value is: 214.40000
```