Pointers

Pointers are variables that hold address of other variables.

Pointers are one of the most distinct and exciting features of C language. They provide power and flexibility.

Benefit of using pointers

- Pointers are more efficient in handling Arrays and Strings.
- Pointers allow passing of function as argument to other functions.
- Pointer code is usually shorter and more efficient.
- Pointers enable dynamic memory management.

Concept of Pointer

Whenever a variable is declared, system will allocate a location to that variable in memory, to hold its value. The location where it is stored has a unique address.

Let us say that system has allocated memory location 80F for variable a.

int a = 10 ;

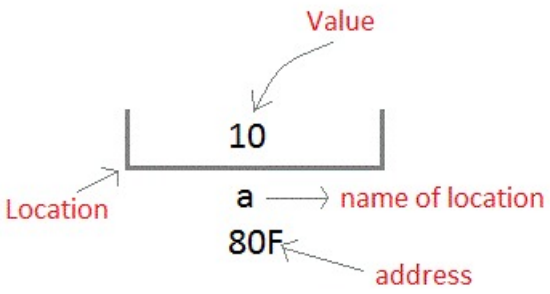

Figure 1: Variable Storage in C

We can access the value 10 by either using the variable name a or the address 80F. Since the memory addresses are simply numbers, they can be assigned to some other variable. Variables that holds memory addresses are called **pointers**. A pointer variable is therefore nothing but a variable that contains an address, which is a location of another variable.

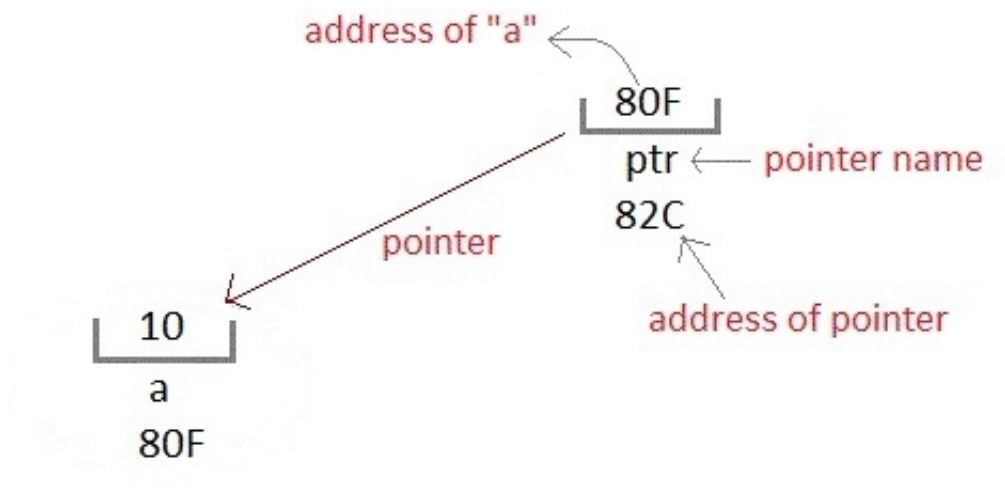

Figure 2: Pointer to Variable

Declaration

General syntax of pointer declaration is:

`data-type* pointer-var;`

Data type of pointer must be same as the variable, which the pointer is pointing. void type pointer works with all data types, but isn't used often.

Initialization

Pointer Initialization is the process of assigning a *valid* address to a pointer variable. The address operator '**&**' is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10;
int* ptr;   //pointer declaration
ptr = &a;   //pointer initialization
```

or as in, **int*** ptr = &a; we can do initialization and declaration together.

A pointer variable should be initialized with the address of a variable of the same datatype. Otherwise there will be unpredictable program behaviour.

```
float a;
int* ptr;
ptr = &a;    //ERROR, type mismatch
```

Dereferencing

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is *dereferenced*, using the indirection operator '*'.

```
#include <stdio.h>
int main() {
    int a;
    int* p;
    a = 10;
    p = &a;
    printf("%d", *p);     //this will print the value of a.
    printf("%d", *&a);    //this will also print the value of a.
    printf("%u", &a);     //this will print the address of a.
    printf("%u", p);      //this will also print the address of a.
    printf("%u", &p);     //this will also print the address of p.
    return 0;
}
```

Example

Program to demonstrate pointer.

```
#include <stdio.h>
int main() {
    int xyz = 10;
    int* ptr;
    ptr = &xyz;
    printf("Value of xyz = %d\n", xyz);
    printf("Address of xyz : %u\n", &xyz);
```



```
8       printf("Pointer ptr has : %u\n", ptr);
9       printf("Value stored in ptr: %d\n", *ptr);
10      return 0;
11  }
```

```
$
$ c99 Program-10-3.c
$ ./a.out
Value of xyz = 10
Address of xyz : 521566740
Pointer ptr has : 521566740
Value stored in ptr: 10
$
```

Figure 3: Pointers and Addresses

Call Mechanisms

There are two ways that arguments can be passed to a function:

- Call by Value
- Call by Reference

By default, C uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Whatever changes made to the values of the arguments are lost when the control exits the function.

Call by Value

The *call by value* method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. Because of this, changes made to the parameter inside the function have no effect on the argument.

The parameter in the called function is initialized with the value of the passed parameter. As long as the parameter has not been declared as constant, the value of the parameter can be changed; but the changes are relevant only within the scope of the called function. They have no effect on the value of the variable in the calling function.

See Figure **??** for understanding the working of the code below:

Example

```
#include <stdio.h>
/* function declaration */
void swap(int x, int y);

int main() {
   /* local variable definition */
    int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values */
    swap(a, b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}

/* function definition to swap the values */
void swap(int x, int y) {
    int t;
    t = x;   /* save the value of x */
    x = y;   /* put y into x        */
    y = t;   /* put t into y        */
    return;
}
```

Call by Reference

The *call by reference* method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect


```
$
$ c99 Program-10-4.c
$ ./a.out
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
$
```

Figure 4: Call by Value Swap

the passed argument.

To pass by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types.

Example

```
#include <stdio.h>
/* function declaration */
void swap(int*, int*);

int main() {
   /* local variable definition */
    int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
   /* calling a function to swap the values.
   * &a indicates pointer to a i.e. address of variable a and
   * &b indicates pointer to b i.e. address of variable b.*/
    swap(&a, &b);
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return 0;
}
/* function definition to swap the values */
```



```
20 void swap(int* x, int* y) {
21     int t;
22     t = *x;     /* save the value at address x */
23     *x = *y;    /* put value at y into x */
24     *y = t;     /* put t into y */
25 }
```

The function swap() is called, the actual values of the varaiables a and b are exchanged because they are passed by reference.

```
$
$ c99 Program-10-5.c
$ ./a.out
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
$
```

Figure 5: Call by Reference Swap

