C Pointers

Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.

Address in C

If you have a variable `var` in your program, `&var` will give you its address in the memory.

We have used address numerous times while using the `scanf()` function.

```
scanf("%d", &var);
```

Here, the value entered by the user is stored in the address of `var` variable. Let's take a working example.

```c
#include <stdio.h>
int main()
{
  int var = 5;
  printf("var: %d\n", var);

  // Notice the use of & before var
  printf("address of var: %p", &var);
  return 0;
}
```

Output

```
var: 5
address of var: 2686778
```

Note: You will probably get a different address when you run the above code.

C Pointers

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

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

You can also declare pointers in these ways.

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

Let's take another example of declaring pointers.

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

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

Assigning addresses to Pointers

Let's take an example.

```c
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 Variable Pointed by Pointers

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

```c
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;
}
```

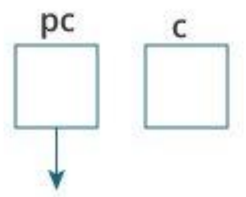
Output

```
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
```

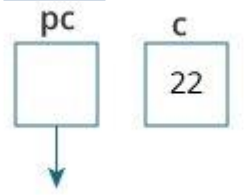
Explanation of the program

1. `int* pc, c;`

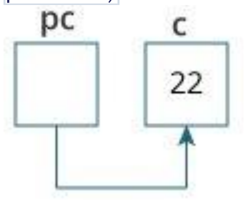
   Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created. Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`

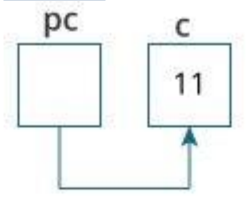
   This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`

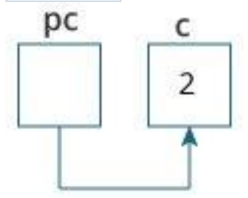
This assigns the address of variable `c` to the pointer `pc`.

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.

Passing Value by Reference

This C program demonstrates the use of pointers to modify the value of a variable through a function call. The main function initializes an integer variable n with the value 2 and prints its initial value. It then calls the function func by passing the address of n as an argument. Inside the func function, the value at the memory location pointed to by the received pointer is incremented by 1. Consequently, when control returns to the main function, the modified value of n is printed, showcasing how the original variable is altered through pointer manipulation. In this way, the program highlights the concept of passing variables by reference using pointers, allowing functions to directly modify the values of variables outside their scope.

```
#include<stdio.h>
void func(int *n)
{
	*n=*n+1;
}
int main() {
	int n=2;
	printf("Value of n before function call: %d", n);
	func(&n);
	printf("\nValue of n after function call: %d", n);
}
```

Output:

```
Value of n before function call: 2

Value of n after function call: 3
```

Iterating an array through a pointer

This C program demonstrates the use of pointers in conjunction with an integer array. The array arr is initialized with values 1 through 5. The pointer ptr is declared and assigned the address of the first element of the array using int *ptr = arr;, effectively making ptr point to the beginning of the array. The program then enters a for loop that iterates five times, each iteration corresponding to an element in the array. Inside the loop, the printf statement is used to display both the value at the memory location pointed to by ptr and the address stored in ptr. The dereference operator (*ptr) retrieves the value at the memory location pointed to by the pointer. After printing, ptr++ increments the pointer, causing it to point to the next element in the array in each iteration. This showcases the concept of pointer arithmetic, where the pointer is automatically adjusted to point to the next location based on the size of the data type it is pointing to. As a result, the loop effectively traverses the entire array, printing the values and addresses of each element. The use of pointers in this program illustrates their role in facilitating efficient and flexible manipulation of data in memory.

```
#include<stdio.h>

int main() {
	int arr[]={1,2,3,4,5};

	int *ptr=arr;

	for(int i=0; i<5; i++)

	{

		printf("\nValue %d  Address: %p  ", *ptr, ptr);

		ptr++;

	}

}
```

In output, you can see the consecutive locations of array elements with a difference of 4 because of the int data type.

Output:

```
Value 1  Address: 000000000065FE00

Value 2  Address: 000000000065FE04

Value 3  Address: 000000000065FE08

Value 4  Address: 000000000065FE0C

Value 5  Address: 000000000065FE10
```