Pointers, Stack & Heap Memory, malloc( )

1 Overview

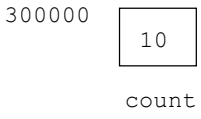
When C was developed, computers were much less powerful and had much less memory to work with. The ability to work directly with particular memory locations was beneficial. Pointers in C allow you to change values passed as arguments to functions, to work with memory that has been dynamically allocated, and to more efficiently work with complex data types, such as large structures, linked lists, and arrays.

2 Pointer Variables

Suppose you declare an integer called `count` as follows:

```
int count = 10;
```

So, in memory, space was reserved for an integer, the name of this space (the variable) is called `count`, and the value of 10 was placed into that memory space. Let's say the address of that location in memory of `count` is `300000`.

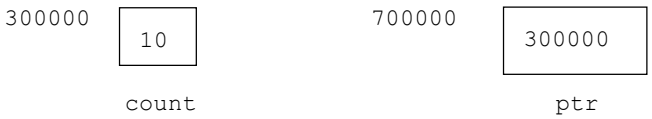


A **pointer** is a ***variable*** whose value is the ***address*** of another variable. You can declare a pointer and have it point to (make its value be) that location in memory for the variable called `count` as follows:

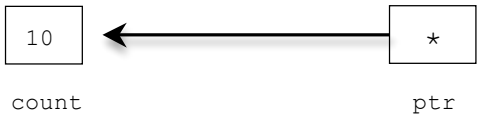
```
int *ptr;
ptr = &count;
```

The asterisk (`*`) in a declaration defines to the system that `ptr` is to be a `pointer` and the `int` is the data type to which it will point. The address operator (`&`) in the second line refers to the address of the item, in this case, the address of `count`, which is of type `int`.

After the first line, let's say the address of that location in memory of where `ptr` is stored is `700000`. After the second line above, `ptr` has the address of `count` as its value (i.e. it points to the location in memory called `count`, which has the value of `10` in it).



We usually represent this graphically as the following:



If you wanted to change the value that `ptr` is pointing to, you can write the following:

```
*ptr = 25;
```

which would have had the same effect as if you had done this:

```
count = 25;
```

In either case, the value at that memory location was changed to `25`. In the first case, where it says

```
*ptr = 25;
```

`ptr` is **dereferenced** by using the asterisk (`*`) because *a direct assignment to a pointer variable will change the address of the pointer*, ***not*** the value at the address that it is pointing to. So, to access the value of what a pointer is pointing to, you have to use the **dereferencing (indirection)** operator, the asterisk (`*`).

So, if you use `ptr` alone in a program, you are referring to the pointer itself.
It you use `*ptr`, then you are referring to what `ptr` points to.

3 Pass-by-Value

Recall that when parameters are passed to functions, the called function makes a copy of the parameter(s). Any changes made to the argument passed in happens only to the copy of the parameter, not to the original argument.

```
void modify (int a) {
     printf ("2. The address of modify's a is %p \n", &a);
     a = 15;
     printf ("3. The value of modify's a is %i \n", a);
}

int main (void) {
     int a = 20;

     printf ("1. The address of main's a is %p \n", &a);
     modify (a);
     printf ("4. The value of main's a is %i \n", a);
     return 0;
}
```

OUTPUT:

```
1. The address of main's a is 0x7fff3d3c8e2c
2. The address of modify's a is 0x7fff3d3c8e0c
3. The value of modify's a is 15
4. The value of main's a is 20
```

4 Pass-by-Reference

For which of the following arguments in the `modify( )` function below will only a local copy be changed?

```
void modify(int p, int *q, int *r) {
    printf ("1. p is %i, q is %i, r is %i \n", p, *q, *r);

    p = 27;
    *q = 27;
    *r = 27;
}

int main (void) {
    int a = 1;
    int b = 1;
    int c = 1;
    int *x = &c;

    modify(a, &b, x);
    printf ("2. a is %i, b is %i, c is %i, x is %i \n", a, b, c, *x); // b and c are changed

    return 0;
}
```

OUTPUT:

```
1. p is 1, q is 1, r is 1
2. a is 1, b is 27, c is 27, x is 27
```

5 Stack and Heap Memory

Two available areas of memory used to store variables are commonly referred to as the ***stack*** **(automatic)** and the ***heap*** **(allocated)**.

Stack resident variables include:

- variables declared inside functions (including the main function), and basic blocks, that are **not** declared *static*
- parameters passed to functions
- (The size of these variables must be known at compile time.)

Stack resident variables are:

- created ("pushed on to") the stack when a basic block that contains them (i.e. function) is activated and
- disappear from ("popped off") the stack when the function is done
- (Those areas of memory that had been used by variables when a basic block was activated may then be overwritten after exit from the block.)
- stack memory is more limited, i.e. there is less of it compared to heap memory

****

Heap resident variables include:

- variables declared outside all functions (globals)
- variables declared inside basic blocks that **are** declared *static*
- memory areas dynamically allocated at run time with *malloc( )*

Storage for declared ***heap resident*** variables is:

- assigned at the time a program is loaded, and
- remains assigned for the life of a program

Storage allocated on the ***heap*** using *malloc( )* remains allocated until:

- either ***free( )*** is called, or
- the program ends

```
#include <stdio.h>

int main (void)
{
    int y;
    int *ptr;
    static int a;

    ptr = &y;
    *ptr = 99;

    printf ("y is %i \nptr is %p \naddress of ptr is %p \n", y, ptr, &ptr);

    ptr = &a;
    printf ("the address of a is %p \n", ptr);

    return 0;
}
```

OUTPUT:

```
y is 99
ptr is 0x7fffca69b39c
address of ptr is 0x7fffca69b390
the address of a is 0x601030
```

← notice this address is far removed from the two above

6 malloc( )

The ***malloc( )*** (memory allocate) function can be used to dynamically allocate an area of memory to be used at run time. Heap memory is used for these variables that use *malloc( )*. Include `<stdlib.h>` whenever using *malloc( )*.

malloc( )

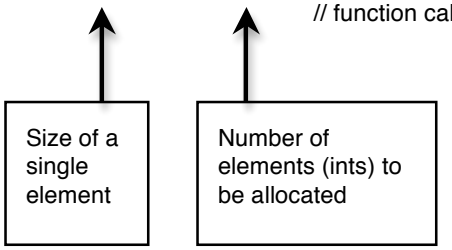
- permits postponing the decision on the size of the memory block needed to store a large data type, such as a large structure or an array;
- permits using a section of memory for the storage of a large data type at one point in time, and then when that memory is no longer needed, it can be freed up for other uses.
- is better when storing large data types, such as a large structure or array, because copying large data structures onto the stack is very inefficient, and
- it's also better because large data structures may result in stack overflow (because there is a finite amount of memory)

The parameter passed to *malloc( )* is the size of the area to be allocated in bytes. Therefore, if you want to allocate space for 100 `ints`, you must pass `400` to *malloc( )* because each `int` occupies 4 bytes. But, the good thing is we can use the ***sizeof*** operator instead of having to know how big each data type is on the particular system we are using.

The *malloc( )* function returns a void pointer, so the type needs to be cast. (If there is not enough memory, it returns NULL.) If you are dynamically allocating space for 100 `ints`, then you would do the following:

```
int *base = NULL;   // declare and initialize an int pointer - - DON'T FORGET TO INITIALIZE POINTERS before use

base = (int *) malloc(sizeof(int) * 100);   // typecast the function call to malloc( ) to also
                                            // be of type int pointer  and assign the result of the
                                            // function call to base (the pointer declared above)
```




```
if (base == NULL){
       fprintf(stderr, "malloc failed \n");
       exit (1);
}


// do stuff ...
...


free(base);
```