Introduction to Pointers

Introduction to Pointer

Pointers are variables whose values are *memory addresses*. Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an *address* of a variable that contains a specific value. In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value

Pointers, like all variables, must be defined before they can be used. Pointer variables are defined with asterisk prefixed with the name of the variable, like shown in here

```
int *ptr ;
```

ptr is now a pointer variable. It is not yet pointing to an address , so it is either having a garbage value or NULL. A pointer with the value NULL points to *nothing.* A pointer may be initialized to NULL, 0 or an address.

```
ptr = NULL ;
```

How do we store address ? Remember, pointers store address of another variable. Then, we will define a new variable, say

```
int x = 100 ;
```

We learnt from scanf function, where we sent the address of a variable using &. The address of x is &x, right ?

Now, make ptr point to variable x by this next statement,

```
ptr =  & x ;
```

The &, or address operator, is a unary operator that returns the address of its operand. In the above statement, we are taking the address of x (using the unary operator & like we used to in the scanf function). Because ptr is a pointer variable , it is perfectly okay to store the address of a variable (in this case x).

But it is not okay to do this

ptr = x ; // why ?

because the value of x (which is just 100) will be stored in ptr. That is, ptr is pointing to an address 100. The compiler won't complain about it, but during runtime, 100 is a memory location (either in the kernel to which we don't have permission to access or 100 may be some random address) which our programs don't have access to and most of time the program crashes. Some of the crashes are mainly the result of accessing a memory location that is not in our space.

Also, note : The operand of the address operator must be a variable; the address operator *cannot* be applied to constants or expressions.

for instance,

ptr = &100 ; // NOT OKAY.

ptr = 100 ; // not OKAY

NOTE: POINTERS CAN ONLY STORE ADDRESS OF ANOTHER VARIABLE.

Because pointers can be made to point to different addresses, we show how pointers can be made to change to point to different memory locations. This is the power of pointers.

To print just address ptr is storing, we say

printf (" The address in ptr %p \n ", ptr) ;

Did you notice %p , a conversion specifier. Also, did you notice use of just pointer variable ptr.

Dereference a Pointer

Dereferencing A Pointer that appears on RHS

We know how to store the address of a variable in a pointer variable.

int *ptr , x = 10 ;

// we declared a pointer variable and a regular variable x = 100

ptr = &x ;

In pointer world, we say pointer ptr is pointing to variable x. Using ptr, we can access the value of x; How ? This is done using the asterisk operator *. It is coincidence that we declare a pointer variable using asterisk and dereference it using an asterisk operator. But they mean differently: one is to declare a pointer variable and the other to mean to dereference.

Deference a pointer variable occurring on the RHS of an equal operator means we are fetching the value the pointer is pointing to. We use the * operator.

For instance ,

say

y = 200 ;

ptr = &y ; // ptr is pointing to y

the statement

x = *ptr ;

means

1. fetch the address stored in the pointer variable ptr, in this case the address of y

2. then, get the value from that address, in this case 200.

This is called dereference.

Dereferencing A Pointer that appears on LHS

Now the supervisor says, can you deliver this letter to this house you are watching.

x = 100;

ptr = &y ; // assignment

*ptr = x ; // dereference on LHS

then the last statement means

store the value of x to where ever ptr is pointing to.

Another example using dereferene operator :

- printf(**"%d", *ptr);**

prints the value of variable y, namely 100.

Here we fetch the value as if it is occurring on the RHS.

- Using * in this manner is called dereferencing a pointer. The operation means different when it occurs on RHS and LHS

Take this example:

y = *ptr + 100 ;

fetching the value where ptr is pointing and add 100.

Other examples ,

y = 100 + *ptr ;

y = *ptr * 2 ;

y = 2**ptr ; // this is not 2 power y like in some languages

This is incorrect

&ptr = &x ;

y = &ptr + 2 ; // this is incorrect unless y is also a pointer.

Here is the video:

Dereferencing Pointers (https://youtu.be/9z-2kUWGWMg)

(https://youtu.be/9z-2kUWGWMg)

Declare and initialize in one statement

Initializing Pointers during declaration

So far we are used to declaring pointers as

int x = 100 ;

int *ptr = &x ;

Instead of declaring a pointer and then assigning an address, we can declare and initialize a pointer in one statement like this

int *ptr = &x ;

This means, we are assigning the address of x to pointer ptr while defining the pointer itself. In this kind of statement, we are completely dealing with addresses.

Alternatively, we could do this too

int x = 100, *ptr = &x ;

Other declarations could be:

int y = 100 ;

int *ptr1 = &y , *ptr2 ;

int x, *ptr3 = &x ;

Now, consider this.

int *ptr4 = ptr1 ; // Watch this out. ptr1 has a memory address , which is the address of a variable y. We are assigning this memory address to ptr4

In other words, ptr1 and ptr4 are pointing to variable y.

but

int *ptr4 = *ptr1 is mistake because *ptr1 is 100. We are assigning address 100 to ptr4 which is incorrect. Right ?

Valid and Invalid pointer assignments

Incorrectly Assigning values to pointers

int *ptr ;

int x ;

1. ptr = 100 ;

would be terribly wrong because the data 100 is stored in the pointer variable. That means, ptr is pointing to the address 100. 100 may be the memory in Kernel space resulting in system error and crashing the program.

2.

x = 1023;

ptr = x ;

again it is wrong. Why ? The value of x is a data value. You cannot store in a pointer variable. ptr would then be pointing to address 1023 which is again in kernel space, not user space.

3.

ptr = x + y ;

again terrible. Why ? you are adding data values of x and y and storing them in a pointer variable.

Here is the video of incorrect assignments of pointer.

4. x = ptr ;

is again wrong. You are taking the address stored in pointer ptr and storing in a regular int variable x. This probably a typo on the part of the programmer, he probably meant

x = *ptr ;

5.

ptr = &x + 10 ;

This is generally okay. Why ? We are getting the address of x and adding 10 to that address. The resulting value is again an address and storing that address in ptr. But avoid this kind of assignment ,

we generally do arithmetic operations with pointers only when they are pointing to arrays or dynamic memory locations, not with variables like this.

6.

*ptr = &x ;

is wrong. Why ? Because we are taking the address of x and storing to a location by dereferencing the ptr.

The programmer may have meant

ptr = &x ; or

*ptr = x ;

Some good examples on how to use pointers

Consider the following statements,

int x = 200, y = 100, *ptr1, *ptr2 ;

we have two variables of type int, two pointer variables of type int.

1.

ptr = &x ;

is perfectly okay. Why ? We are taking the address of x and storing it in the pointer variable. In other words, ptr is now pointing to x.

2.

*ptr = x ;

is good. We are placing the value of x in the location ptr is pointing. But make sure ptr is pointing to a valid location. If ptr is not pointing to any location, you will runtime error.

3.

y = *ptr ;

is good too. Again, make sure ptr is pointing to a valid memory location

4.

*ptr = x + y ;

is good. We are adding x and y, copying the sum to a location where ptr is pointing . Again, make sure ptr is pointing to.

5.

x = *ptr + 5 ; is good too.

Pointer Arithmetic with Arrays

Consider this array

char cData[4] = { 'A', 'B', 'C', 'D' } ;

Here cData is called pointer constant. cData is also the array name pointing to the first cell.

NOTE: Though cData is also an array, it is also a pointer constant.

you cannot change cData to point elsewhere. It is stuck pointing to the first cell.

Because array name is acting like a pointer, we can assign the array to a pointer too. Like shown here

char *ptr = cData ;

is a perfectly valid statement and ptr is pointing to cData.

Deferencing it

*ptr would yield 'A'.

Now, when we write

ptr = ptr + 1 ;

We are adding one to the address in ptr. Basically we are incrementing or advancing the pointer to the next address. In this case , ptr is now pointing to the next cell in the array.

dereferencing *ptr would then fetch 'B' for us.

The above statement can also be written as

ptr++ ; // is same as ptr = ptr + 1 .

ptr++ would then advance to the next cell.

*ptr would then fetch 'C'

ptr++ one more time would point to the next cell

*ptr would fetch 'D'

Consider this array

int iData [4] = { 0x1, 0x2, 0x3, 0x4 } ;

say the address of

first cell = 4400

second = 4404

third cell = 4408

fourth cell = 4412

int *iPtr = iData ; // point iPtr to the address 4400..

Here iPtr is pointing to iData

iPtr++ ;

would advance the pointer to the second cell.

iPtr++ would advance to the third cell.

So, when it comes to pointer arithmetic, say ptr + 1 is not just add one to the pointer. The resulting value is based on the type of the pointer times the number.

For instance, say ptr is pointing to an address 4400 like above.

ptr + x is

ptr + (x times sizeof (datatype of pointer)) = ptr + (x * sizeof (int))

For ptr + 1 = it is ptr + (1 * 4) = 4400 + 4 = 4404 which is the address of second cell

ptr + 2 = ptr + (2 * 4) = 4400 + 8 = 4408 is the address of the third cell

You can take this example and run it yourself

```
#include <stdio.h>
int main ( )
{
        int *iPtr ;
        int iData[4] = {1, 2, 3, 4};
        iPtr = iData;
         printf("before %p\n", iPtr);
         iPtr = iPtr + 1;
         printf("After %p\n", iPtr);
}
```

Dynamic Memory Allocation functions: malloc, free, calloc, realloc, memove

Memory can be allocated from two pools: stack and Heap. When we declared variables in a function, we allocated memory from the stack space.

when we declare say a variable

int x ;

The four bytes is allocated from the stack , predetermined during compile time.

But, programmers may not know how much memory the program requires when they are writing the program. So , during runtime, we need memory to be allocated. Also, Self referential structure require to be allocated dynamically. malloc is one system function that you would call to allocate memory. This memory is allocated from heap space.

Step 1:

The sysetm function to allocate memory during runtime is malloc

int x= 8;

malloc (x) ; // this allocates 8 bytes

malloc (x+4) ; // allocates 12 bytes

malloc (1024) ; // allocates 1024 bytes

STEP 2 :

When the malloc returns, it returns a void pointer to the memory allocated. We cannot traverse the memory using a void pointer. So we need to cast the memory allocated to one of the known types: int, char, short, or any user defined structure.

Say we need to traverse the memory using a int.

We do this by casting the void pointer to a int pointer like shown.

```
int *ptr ;

ptr = ( int  * ) malloc ( 2* sizeof ( int)  ) ;
```

In the above statement, the void pointer that is returned by malloc, is type casted to int * pointer. Then, we assign it to ptr which is int *.

In short, the above line does three things, all at once:

1. allocate a memory of 8 bytes = 2 times sizeof (int)
2. cast the memory to int *
3. make a int pointer ptr point to that memory

Because we allocated 2 int (= 8 bytes) , we can say ptr is now pointing to the first int (4 bytes) .

Now, we can assign a value to the ptr.

```
*ptr = 0x0a0b0c0d ;
```

which assigns 0x0a0b0c0d to the first int.

When we do ++ptr, we have advanced the pointer to the second int.

Now we assign

```
*ptr = 0xd0c0b0a ;
```

this means we have assigned 0xd0c0b0a to the *ptr which in this case it is pointing to the second int.

Though we explained with just two ints, in real life, the number of memory allocated is vastly huge.

Just to summarize: When we allocate memory using malloc, we get the memory from the heap, not from stack.

malloc int pointer (https://www.youtube.com/watch?v=896nrvtA_xE&feature=youtu.be)

(https://www.youtube.com/watch?v=896nrvtA_xE&feature=youtu.be)

Let us take another example.

```
struct _user {
    char name [ 12 ] ;
    int age ;
} ;
```

let us declare a structure pointer of type

```
struct    _user     *sPtr ;
```

Now we will allocate memory whose size is twice the size of a structure _user.

```
sPtr =  ( struct   _user   *  )  malloc  ( 2 * sizeof (  struct    _user  ) ) ;
```

now sPtr is now pointing to the dynamic memory whose size is twice the size of the structure.

sPtr is pointing to the first structure and we could assign values such as

sPtr->age = 10 ;

strcpy (sPtr->name, "John");

How do I advance the pointer to the next structure?

sPtr++ would make it point to the next cell, we could assign the values

sPtr->age = 20

strcpy (sPtr->name, "Jenny");

If we do sPtr++ one more time, the sPtr would be pointing beyond the allocated memory. When you do this, you may end up in a crash. So, it is programmers responsibility to make sure the pointers are within the allocated memory size.

malloc a structure (https://youtu.be/Rnwt--h7x7Y)

(https://youtu.be/Rnwt--h7x7Y)

memove :

#include <string.h>

void *memmove(void *dest, const void *src, size_t n);

The memmove() function copies n bytes from memory area src to memory area dest. The memory areas may overlap: copying takes place as though the bytes in src are first copied into a temporary array that does not overlap src or dest, and the bytes are then copied from the temporary array to dest.

calloc:

void *calloc(size_t nmemb, size_t size);

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

realloc:

void *realloc(void *ptr, size_t size);

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

Here is the sample program :

/// (file) **dynamic.c (https://csus.instructure.com/courses/94454/files/14915339/download?wrap=1)** ⬇ (https://csus.instructure.com/courses/94454/files/14915339/download?download_frd=1)

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

#define MAX_SIZE 32
#define INCREASE_SIZE 10

int main ( void )
{

    unsigned char *src ;
    unsigned char *dest ;
    src = ( unsigned char * ) malloc ( MAX_SIZE ) ;
```

```
    dest = ( unsigned char * ) malloc ( MAX_SIZE ) ;

     memset ( src, 255, MAX_SIZE);  // assign 255 to cells

     memmove ( dest, src, MAX_SIZE);   // copy the values to dest pointer

     int i = 0;
     for ( ; i < MAX_SIZE ; i++ )   {
          printf ( "value at i=%d is %d \n", i, dest[i]);
     }

      dest = ( unsigned char * ) realloc ( dest, MAX_SIZE+INCREASE_SIZE );  // INCREASE
      memset ( dest + MAX_SIZE,  64,    INCREASE_SIZE);  // initialize with 64

      for (    ;    i < MAX_SIZE+INCREASE_SIZE   ;    i++ )    {
              printf ( "value at i=%d is %d \n", i, dest[i]);   // printing the whole dest
      }
    // two ways to free the memory.  The standard one is:
     free ( src ) ;


}
```