Pointers

A pointer is a variable that stores the address of another variable.  
A pointer indirectly references a value.   
Referencing a value through a pointer is called indirection.

&- reference operator  
\*-Dereferencing operator

##### EXAMPLE:

Given sizeof(int) = 2 bytes

| # | Address | Value |
| --- | --- | --- |
| x | 6048 | 517 |
| y | 7096 | 6048 |
| z | 8016 | 7096 |
| a[0] | 65000 | 8016 |
| a[10] | 65020 | 7096 |

| y | 6048 |
| --- | --- |
| z | 7096 |
| a[4] | N.D (not 8008) |
| a[0] | 8016 |
| x | 517 |

##### "Address of" (&) operator - reference operator

##### The address of the operator is used to give or produce the memory address of a data variable.

Example from above,

| &a[5] | 65010 |
| --- | --- |
| &x | 6048 |
| &z | 8016 |
| &a | 65000 |
| &y | 7096 |
| &&x | Error! |

&x = 6048 but it isn’t == value of y. They are different types altogether.

&(&x) is asking the address of an address

Hence an error will be encountered

##### 

##### "Value of address" (\*) operator - Dereferencing operator

The value at the address or \* operator is used to obtain the value present at a given memory address. It is denoted by \*

| \*y | 517 |
| --- | --- |
| \*\*z | 517 |
| \*\*\*a[0] | 517 |
| \*z | 6048 |
| \*a[10] | 6048 |

Mixed examples

| \*x | Garbage/Not Defined |
| --- | --- |
| \*(&x) | 517 |
| &(\*z) | Here, we’re addressing a value (\*z) i.e 6048,  (which is not a variable) which gives us an error. |

Format specifier for printing pointer address is ‘%u’

*int a=10 , b=20;  
int \*p1, \*p2;  
int \*\*p;*

*p1 = &a;  
p2 = &b;*

*p = &p2;*

*\*p1 = \*p1 + \*p2; \\ \*p1 is a(10), \*p2 is b(20).Therefore, Now \*p1=a=30.*

*\*p2 = \*p1 - \*\*p; \\\*p1 is 30, \*\*p is 20. Therefore, Now \*p2=b=10.*

*\*p1 = a - b; \\a is 30, b is 10. \*p1 = a = 20.*

|  |
| --- |

##### Pointers to a pointer:

Just like p2 is pointing to a or storing the address of b, we can have another variable p which can store the address of p2.

int \*\*p;  
p = &p2;

##### 

##### Types of function calls

Based on the way we pass arguments to the function, function calls are of two types.

1. Call by value --> sending the values of arguments

Here the values of the arguments are passed to the function.   
Consider this example:  
int c = sum ( x , y ); => Assume x=3 and y=4  
If the sum is defined as sum(int a, int b), the values 3 and 4 are copied to a and b. Now even if we change a and b, nothing happens to the variables x and y.  
This is ‘call by value’.  
In C we usually make a call by value.

1. Call by reference --> sending the address of arguments

Here the address of the variable is passed to the function as arguments.  
Now since the addresses are passed to the function,   
the function can now modify the value of a variable in the calling function using \* and & operators.

Example:  
*void swap(int \*x, int \*y)  
{  
 int temp;  
 temp= \*x;  
 \*x = \*y;  
 \*y = temp;  
}*

This function is capable of swapping the values passed to it.   
If a=3 and b=4, before a call to swap(a,b), a=4 and b=3 after calling swap.

*int main()*

*{*

*int a=3; // a is 3 and b is 4*

*int b=4;*

*swap(&a, &b)*

*return 0; // now a is 4 and b is 3*

*}*

Types of Pointers

* Void Pointer

A pointer that has no associated data type with it.  
It can point to any data of any data type.  
It cannot be referenced directly (\* cannot be used directly )   
since its length is always undetermined.

But, they can be typecasted to any type.

Uses of the void pointer,  
malloc and calloc functions return a void pointer.   
Due to this reason, they can allocate memory for any type of data.

Example,

*int main  
{  
 int n = 10;*

*void \*ptr = &n;*

*printf ( “%d” , \*( int\* )ptr ); //Typecasting to int type.  
 // printf(“%d”, \*ptr ); We cannot dereference a void pointer.  
 hence, error.*

*return 0;  
}*

* Null Pointer

A pointer that does not point to any memory location.  
It represents an invalid memory location.

Used to initialize a pointer when it isn’t provided any valid memory address yet.  
Useful for handling errors while using the malloc function.  
  
Size of null pointers = Size of normal pointers (Depends on the platform)

Good practices to-   
1. To initialize pointers to NULL.

2. NULL check before dereferencing any pointer to avoid surprises.

-------------------------------------------------------------------

Pointers play a very important role in dynamic memory allocation.  
Allocated memory can only be accessed through pointers.

Continued to Dynamic Memory….