Some C programming tasks are performed more easily with pointers, and other tasks such as dynamic memory allocation, cannot be performed without using pointers. So, it becomes necessary to learn pointers.

As you know, every variable has a memory location and every memory location has its address defined which is accessed using ampersand & or address of operator, which physically denotes an address in memory.

A **pointer** is a variable that contains the address of another variable. Just like any variable or data constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * is used to declare a pointer and is the same asterisk used for multiplication. However, in this declaration statement the asterisk is being used to designate a variable as a pointer. Here are some valid pointer declarations:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *cp     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable that the pointer points to.

There are a few important operations that are related to pointers and are performed frequently:

(a) define a pointer variable
(b) assign the address of a variable to a pointer
(c) access the value at the address stored in the pointer variable. This is accomplished by using unary ***** operator.

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **NULL** pointer.

```
int *ptr = NULL;
```

The NULL pointer is a constant with a value of zero and is defined in several standard libraries.

In most operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location and by convention it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Pointers have many applications that are very important to C programming. The following pointer concepts should be clearly understood by any C programmer:

Concept & Description
Pointer arithmetic There are four arithmetic operators that can be used in pointers: ++, --, +, -
Passing pointers to functions in C Passing an argument by reference/address enable the passed argument to be changed within the calling function

A pointer in C is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just like you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes in size next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character is one byte in size and will be available at 1001.

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The C programming language also allows passing a pointer to a function. This basically changes the formal parameters from being an input data value to an output where the function can now return multiple values back to the calling function. To do so, simply declare the function parameter as a pointer type.

```
void ptr_ex (int *ptr);
```

In the above function prototype, the *ptr field will only accept the address of an integer, such as &num where num was declared as an integer in the calling function. When passing an address, this is called passing by reference. The data will be stored in the calling function’s declared variable num.

```
ptr_ex(&num);
```

Inside the function definition, the data is sent back by dereferencing. Dereferencing is a process that stores or retrieves the data from the calling function.

```
*ptr = *ptr + 10;
```

Which can be analyzed with the passed address and rewritten as:

```
*&num = *&num + 10;
```

The *& can be thought of a cancelling operation where num is where the data is retrieved from and then stored to.

```
num = num + 10;
```

Applied Pointer Examples
The address of operator & has been used numerous times when calling the scanf() function.

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

Every variable is associated with both a value and address in memory.

```
#include <stdio.h>
void main(void)
{
  int var = 5;
  printf("var: %d\n", var);
  printf("address of var: %p", &var);  // Notice the use of & before var
}
```

```
Program Output
var: 5
address of var: 2686778
```

Note: You will most certainly get a different address when you run the above code due to the installed programs, computer options, and available memory in your computer.

Pointers (pointer variables) are special variables that are used to store addresses of specific variable types rather than data values. When declaring pointer variables, the unary * operator is used to define the variable as a pointer.

Various methods of declaring a pointer variable:

```
int* p;
int *p1; // preferred method
int * p2;
int *p1, p2;
```

The first 3 examples declare a single pointer using various * operator spacing, the last example declares p1 as a pointer and p2 as regular variable which is legal.

Pointer variables CANNOT be used when uninitialized, so a viable data type address must be assigned to it before it can be used.

```
int *pc, c;
c = 5;
pc = &c;
```

Here, integer 5 is assigned to variable c and the address of integer variable c is assigned to pointer pc. Notice that once the pointer has been declared, the * operator is omitted when assigning the address to pointer variable pc.

Now that the pointer has a viable address stored, it can be used to point to or be dereferenced, to retrieve or store the data pointed to at its address. This is where most of the pointer confusion comes from. The * operator has a different meaning when used inside of code statements than when it was used to declare the pointer. The * operator dereferences the address of operator so the pointer will now go to the address it has stored, or to the variable name of the address stored.

```
int *pc, c;
c = 5;
pc = &c;
printf("%d", *pc);
```

When the code snippet on the previous page is run, the printf("%d", *pc); will output a 5 to the screen. A simple way to visualize this dereferencing operation is to look at what is stored in pc -> pc = &c. If we rewrite this with the * operator inserted on both sides, *pc = *&c it is easy to visualize that the *& operators cancel each other out so *pc is actually variable c as far as the program is concerned. Therefore, the printf() is printing what is stored in c.

The below example shows the variable data can change which has no effect on where the pointer points.

```
int *pc, c;
```

```
c = 5;
pc = &c;
c = 1;
printf("%d", c);    // prints 1
printf("%d", *pc);  // prints 1
```

Or we can use the pointer to change the variable data as well.

```
int *pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);  // prints 1
printf("%d", c);    // prints 1
```

If we rewrite the code line containing *pc = 1 using the line of code above it, *&c = 1 which when dereferenced means c is assigned the value of 1.

Pointers can also have the stored address value changed to point to a different variable of the same type.

```
int* pc, c, d;
c = 5;
d = -15;
pc = &c;
printf("%d", *pc); // prints 5
pc = &d;
printf("%d", *pc); // prints -15
```

The primary reason for using pointers is to use a process called Call by Reference. This syntax structure allows any function to have multiple outputs if the programmer desires and is accomplished by passing an address as the actual parameter, or argument, to the function.

```
#include <stdio.h>
void swap (int *n1, int *n2); // notice both formal parameters are declaring pointers
void main(void)
{
    int num1 = 5, num2 = 10;
    swap( &num1, &num2); // address of num1 and num2 is passed as actual parameters
    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
}
void swap(int* n1, int* n2)  // actual passed parameters are &num1, &num2
{
    int temp;
    temp = *n1; // temp = *&num1 or the value in num1 is assigned to temp
    *n1 = *n2; // *&num1 = *&num2 or the value in num2 is assigned to num1
    *n2 = temp; // *&num2 = temp or the value in num1 is assigned to numb2
}
```

Program Output:
num1 = 10
num2 = 5

The addresses of num1 and num2 are passed as actual parameters to the swap() function using swap(&num1, &num2);. The declared formal parameters of int *n1 and int *n2 are declaring n1 and n2 as pointers which will accept these address arguments in the function definition.

This technique is known as call by reference in C programming.