Pointers in C are easy and fun to learn. 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 to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

[Live Demo](http://tpcg.io/SxljuD)

#include <stdio.h>

int main () {

int var1;

char var2[10];

printf("Address of var1 variable: %x\n", &var1 );

printf("Address of var2 variable: %x\n", &var2 );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or 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 \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

int \*ip; /\* pointer to an integer \*/

double \*dp; /\* pointer to a double \*/

float \*fp; /\* pointer to a float \*/

char \*ch /\* 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 or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

[Live Demo](http://tpcg.io/Lt9V7y)

#include <stdio.h>

int main () {

int var = 20; /\* actual variable declaration \*/

int \*ip; /\* pointer variable declaration \*/

ip = &var; /\* store address of var in pointer variable\*/

printf("Address of var variable: %x\n", &var );

/\* address stored in pointer variable \*/

printf("Address stored in ip variable: %x\n", ip );

/\* access the value using the pointer \*/

printf("Value of \*ip variable: %d\n", \*ip );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

NULL Pointers

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.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

[Live Demo](http://tpcg.io/xGfUyr)

#include <stdio.h>

int main () {

int \*ptr = NULL;

printf("The value of ptr is : %x\n", ptr );

return 0;

}

When the above code is compiled and executed, it produces the following result −

The value of ptr is 0

In most of the 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. But by convention, if a pointer contains the null (zero) value, 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 \*/

# C - Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as 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 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 will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

[Live Demo](http://tpcg.io/TRQKAE)

#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};

int i, \*ptr;

/\* let us have array address in pointer \*/

ptr = var;

for ( i = 0; i < MAX; i++) {

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, \*ptr );

/\* move to the next location \*/

ptr++;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var[0] = bf882b30

Value of var[0] = 10

Address of var[1] = bf882b34

Value of var[1] = 100

Address of var[2] = bf882b38

Value of var[2] = 200

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

[Live Demo](http://tpcg.io/czaLm5)

#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};

int i, \*ptr;

/\* let us have array address in pointer \*/

ptr = &var[MAX-1];

for ( i = MAX; i > 0; i--) {

printf("Address of var[%d] = %x\n", i-1, ptr );

printf("Value of var[%d] = %d\n", i-1, \*ptr );

/\* move to the previous location \*/

ptr--;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var[2] = bfedbcd8

Value of var[2] = 200

Address of var[1] = bfedbcd4

Value of var[1] = 100

Address of var[0] = bfedbcd0

Value of var[0] = 10

Pointer Comparisons

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 following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

[Live Demo](http://tpcg.io/rqG9AD)

#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};

int i, \*ptr;

/\* let us have address of the first element in pointer \*/

ptr = var;

i = 0;

while ( ptr <= &var[MAX - 1] ) {

printf("Address of var[%d] = %x\n", i, ptr );

printf("Value of var[%d] = %d\n", i, \*ptr );

/\* point to the previous location \*/

ptr++;

i++;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var[0] = bfdbcb20

Value of var[0] = 10

Address of var[1] = bfdbcb24

Value of var[1] = 100

Address of var[2] = bfdbcb28

Value of var[2] = 200

# C - Array of pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers −

[Live Demo](http://tpcg.io/g9VARy)

#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};

int i;

for (i = 0; i < MAX; i++) {

printf("Value of var[%d] = %d\n", i, var[i] );

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer −

int \*ptr[MAX];

It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows −

[Live Demo](http://tpcg.io/UJoygb)

#include <stdio.h>

const int MAX = 3;

int main () {

int var[] = {10, 100, 200};

int i, \*ptr[MAX];

for ( i = 0; i < MAX; i++) {

ptr[i] = &var[i]; /\* assign the address of integer. \*/

}

for ( i = 0; i < MAX; i++) {

printf("Value of var[%d] = %d\n", i, \*ptr[i] );

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

You can also use an array of pointers to character to store a list of strings as follows −

[Live Demo](http://tpcg.io/UJBnBq)

#include <stdio.h>

const int MAX = 4;

int main () {

char \*names[] = {

"Zara Ali",

"Hina Ali",

"Nuha Ali",

"Sara Ali"

};

int i = 0;

for ( i = 0; i < MAX; i++) {

printf("Value of names[%d] = %s\n", i, names[i] );

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of names[0] = Zara Ali

Value of names[1] = Hina Ali

Value of names[2] = Nuha Ali

Value of names[3] = Sara Ali

# C - Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int −

int \*\*var;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

[Live Demo](http://tpcg.io/p3HH8m)

#include <stdio.h>

int main () {

int var;

int \*ptr;

int \*\*pptr;

var = 3000;

/\* take the address of var \*/

ptr = &var;

/\* take the address of ptr using address of operator & \*/

pptr = &ptr;

/\* take the value using pptr \*/

printf("Value of var = %d\n", var );

printf("Value available at \*ptr = %d\n", \*ptr );

printf("Value available at \*\*pptr = %d\n", \*\*pptr);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of var = 3000

Value available at \*ptr = 3000

Value available at \*\*pptr = 3000

# Passing pointers to functions in C

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

[Live Demo](http://tpcg.io/VJgfOL)

#include <stdio.h>

#include <time.h>

void getSeconds(unsigned long \*par);

int main () {

unsigned long sec;

getSeconds( &sec );

/\* print the actual value \*/

printf("Number of seconds: %ld\n", sec );

return 0;

}

void getSeconds(unsigned long \*par) {

/\* get the current number of seconds \*/

\*par = time( NULL );

return;

}

When the above code is compiled and executed, it produces the following result −

Number of seconds :1294450468

The function, which can accept a pointer, can also accept an array as shown in the following example −

[Live Demo](http://tpcg.io/EVWfwS)

#include <stdio.h>

/\* function declaration \*/

double getAverage(int \*arr, int size);

int main () {

/\* an int array with 5 elements \*/

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

/\* pass pointer to the array as an argument \*/

avg = getAverage( balance, 5 ) ;

/\* output the returned value \*/

printf("Average value is: %f\n", avg );

return 0;

}

double getAverage(int \*arr, int size) {

int i, sum = 0;

double avg;

for (i = 0; i < size; ++i) {

sum += arr[i];

}

avg = (double)sum / size;

return avg;

}

When the above code is compiled together and executed, it produces the following result −

Average value is: 214.40000

# Return pointer from functions in C

We have seen in the last chapter how C programming allows to return an array from a function. Similarly, C also allows to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example −

int \* myFunction() {

.

.

.

}

Second point to remember is that, it is not a good idea to return the address of a local variable outside the function, so you would have to define the local variable as **static** variable.

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

[Live Demo](http://tpcg.io/cWvLL4)

#include <stdio.h>

#include <time.h>

/\* function to generate and return random numbers. \*/

int \* getRandom( ) {

static int r[10];

int i;

/\* set the seed \*/

srand( (unsigned)time( NULL ) );

for ( i = 0; i < 10; ++i) {

r[i] = rand();

printf("%d\n", r[i] );

}

return r;

}

/\* main function to call above defined function \*/

int main () {

/\* a pointer to an int \*/

int \*p;

int i;

p = getRandom();

for ( i = 0; i < 10; i++ ) {

printf("\*(p + [%d]) : %d\n", i, \*(p + i) );

}

return 0;

}

When the above code is compiled together and executed, it produces the following result −

1523198053

1187214107

1108300978

430494959

1421301276

930971084

123250484

106932140

1604461820

149169022

\*(p + [0]) : 1523198053

\*(p + [1]) : 1187214107

\*(p + [2]) : 1108300978

\*(p + [3]) : 430494959

\*(p + [4]) : 1421301276

\*(p + [5]) : 930971084

\*(p + [6]) : 123250484

\*(p + [7]) : 106932140

\*(p + [8]) : 1604461820

\*(p + [9]) : 149169022