**C Pointers**

In this tutorial, you'll learn about pointers; what pointers are, how do you use them and the common mistakes you might face when working with them with the help of examples.

Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.

**Address in C**

If you have a variable var in your program, &var will give you its address in the memory.

We have used address numerous times while using the scanf() function.

scanf("%d", &var);

Here, the value entered by the user is stored in the address of var variable. Let's take a working example.

#include <stdio.h>

int main()

{

int var = 5;

printf("var: %d\n", var);

// Notice the use of & before var

printf("address of var: %p", &var);

return 0;

}

**Output**

var: 5

address of var: 2686778

**Note:** You will probably get a different address when you run the above code.

**C Pointers**

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

**Pointer Syntax**

Here is how we can declare pointers.

int\* p;

Here, we have declared a pointer p of int type.

You can also declare pointers in these ways.

int \*p1;

int \* p2;

Let's take another example of declaring pointers.

int\* p1, p2;

Here, we have declared a pointer p1 and a normal variable p2.

**Assigning addresses to Pointers**

Let's take an example.

int\* pc, c;

c = 5;

pc = &c;

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

**Get Value of Thing Pointed by Pointers**

To get the value of the thing pointed by the pointers, we use the \* operator. For example:

int\* pc, c;

c = 5;

pc = &c;

printf("%d", \*pc); // Output: 5

Here, the address of c is assigned to the pc pointer. To get the value stored in that address, we used \*pc.

**Note:**In the above example, pc is a pointer, not \*pc. You cannot and should not do something like \*pc = &c;

By the way, \* is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

**Changing Value Pointed by Pointers**

Let's take an example.

int\* pc, c;

c = 5;

pc = &c;

c = 1;

printf("%d", c); // Output: 1

printf("%d", \*pc); // Ouptut: 1

We have assigned the address of c to the pc pointer.

Then, we changed the value of c to 1. Since pc and the address of c is the same, \*pc gives us 1.

Let's take another example.

int\* pc, c;

c = 5;

pc = &c;

\*pc = 1;

printf("%d", \*pc); // Ouptut: 1

printf("%d", c); // Output: 1

We have assigned the address of c to the pc pointer.

Then, we changed \*pc to 1 using \*pc = 1;. Since pc and the address of c is the same, c will be equal to 1.

Let's take one more example.

int\* pc, c, d;

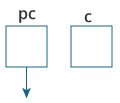
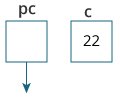
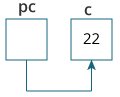
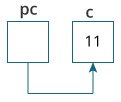
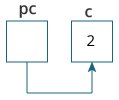
c = 5;

d = -15;

pc = &c; printf("%d", \*pc); // Output: 5

pc = &d; printf("%d", \*pc); // Ouptut: -15

**Explanation of the program**

1. int\* pc, c;  
     
   Here, a pointer pc and a normal variable c, both of type int, is created.  
   Since pc and c are not initialized at initially, pointer pc points to either no address or a random address. And, variable c has an address but contains random garbage value.
2. c = 22;  
     
   This assigns 22 to the variable c. That is, 22 is stored in the memory location of variable c.
3. pc = &c;  
     
   This assigns the address of variable c to the pointer pc.
4. c = 11;  
     
   This assigns 11 to variable c.
5. \*pc = 2;  
     
   This change the value at the memory location pointed by the pointer pc to 2.

### Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,

int c, \*pc;

// pc is address but c is not

pc = c; // Error

// &c is address but \*pc is not

\*pc = &c; // Error

// both &c and pc are addresses

pc = &c; // Not an error

// both c and \*pc values

\*pc = c; // Not an error

Here's an example of pointer syntax beginners often find confusing.

#include <stdio.h>

int main() {

int c = 5;

int \*p = &c;

printf("%d", \*p); // 5

  return 0;

}

**Why didn't we get an error when using int \*p = &c;?**

It's because

int \*p = &c;

is equivalent to

int \*p:

p = &c;

In both cases, we are creating a pointer p (not \*p) and assigning &c to it.

To avoid this confusion, we can use the statement like this:

int\* p = &c;

**Relationship Between Arrays and Pointers**

In this tutorial, you'll learn about the relationship between arrays and pointers in C programming. You will also learn to access array elements using pointers.

Before you learn about the relationship between arrays and pointers, be sure to check these two topics:

* [C Arrays](https://www.programiz.com/c-programming/c-arrays)
* [C Pointers](https://www.programiz.com/c-programming/c-pointers)

**Relationship Between Arrays and Pointers**

An array is a block of sequential data. Let's write a program to print addresses of array elements.

#include <stdio.h>

int main() {

int x[4];

int i;

for(i = 0; i < 4; ++i) {

printf("&x[%d] = %p\n", i, &x[i]);

}

printf("Address of array x: %p", x);

return 0;

}

**Output**

&x[0] = 1450734448

&x[1] = 1450734452

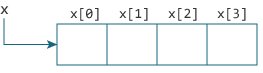
&x[2] = 1450734456

&x[3] = 1450734460

Address of array x: 1450734448

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).

Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.

Relation between Arrays and Pointers

From the above example, it is clear that &x[0] is equivalent to x. And, x[0] is equivalent to \*x.

Similarly,

* &x[1] is equivalent to x+1 and x[1] is equivalent to \*(x+1).
* &x[2] is equivalent to x+2 and x[2] is equivalent to \*(x+2).
* ...
* Basically, &x[i] is equivalent to x+i and x[i] is equivalent to \*(x+i).

**Example 1: Pointers and Arrays**

#include <stdio.h>

int main() {

int i, x[6], sum = 0;

printf("Enter 6 numbers: ");

for(i = 0; i < 6; ++i) {

// Equivalent to scanf("%d", &x[i]);

scanf("%d", x+i);

// Equivalent to sum += x[i]

sum += \*(x+i);

}

printf("Sum = %d", sum);

return 0;

}

When you run the program, the output will be:

Enter 6 numbers: 2

3

4

4

12

4

Sum = 29

Here, we have declared an array x of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that **pointers and arrays are not the same**.

There are a few cases where array names don't decay to pointers. To learn more, visit: [When does array name doesn't decay into a pointer?](https://stackoverflow.com/questions/17752978/exceptions-to-array-decaying-into-a-pointer)

### Example 2: Arrays and Pointers

#include <stdio.h>

int main() {

int x[5] = {1, 2, 3, 4, 5};

int\* ptr;

// ptr is assigned the address of the third element

ptr = &x[2];

printf("\*ptr = %d \n", \*ptr); // 3

printf("\*(ptr+1) = %d \n", \*(ptr+1)); // 4

printf("\*(ptr-1) = %d", \*(ptr-1)); // 2

return 0;

}

When you run the program, the output will be:

\*ptr = 3

\*(ptr+1) = 4

\*(ptr-1) = 2

In this example, &x[2], the address of the third element, is assigned to the ptr pointer. Hence, 3 was displayed when we printed \*ptr.

And, printing \*(ptr+1) gives us the fourth element. Similarly, printing \*(ptr-1) gives us the second element.

# C Pass Addresses and Pointers

In this tutorial, you'll learn to pass addresses and pointers as arguments to functions with the help of examples.

In C programming, it is also possible to pass addresses as arguments to functions.

To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses. Let's take an example:

## Example: Pass Addresses to Functions

#include <stdio.h>

void swap(int \*n1, int \*n2);

int main()

{

int num1 = 5, num2 = 10;

// address of num1 and num2 is passed

swap( &num1, &num2);

printf("num1 = %d\n", num1);

printf("num2 = %d", num2);

return 0;

}

void swap(int\* n1, int\* n2)

{

int temp;

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

}

When you run the program, the output will be:

num1 = 10

num2 = 5

The address of num1 and num2 are passed to the swap() function using swap(&num1, &num2);.

Pointers n1 and n2 accept these arguments in the function definition.

void swap(int\* n1, int\* n2) {

... ..

}

When \*n1 and \*n2 are changed inside the swap() function, num1 and num2 inside the main() function are also changed.

Inside the swap() function, \*n1 and \*n2 swapped. Hence, num1 and num2 are also swapped.

Notice that swap() is not returning anything; its return type is void.

## Example 2: Passing Pointers to Functions

#include <stdio.h>

void addOne(int\* ptr) {

(\*ptr)++; // adding 1 to \*ptr

}

int main()

{

int\* p, i = 10;

p = &i;

addOne(p);

printf("%d", \*p); // 11

return 0;

}

Here, the value stored at p, \*p, is 10 initially.

We then passed the pointer p to the addOne() function. The ptr pointer gets this address in the addOne() function.

Inside the function, we increased the value stored at ptr by 1 using (\*ptr)++;. Since ptr and p pointers both have the same address, \*p inside main() is also 11.

# C Dynamic Memory Allocation

In this tutorial, you'll learn to dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc().

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

## C malloc()

The name "malloc" stands for memory allocation.

The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a [pointer](https://www.programiz.com/c-programming/c-pointers) of void which can be casted into pointers of any form.

### Syntax of malloc()

ptr = (castType\*) malloc(size);

**Example**

ptr = (float\*) malloc(100 \* sizeof(float));

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

The expression results in a NULL pointer if the memory cannot be allocated.

## C calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.

### Syntax of calloc()

ptr = (castType\*)calloc(n, size);

**Example:**

ptr = (float\*) calloc(25, sizeof(float));

The above statement allocates contiguous space in memory for 25 elements of type float.

## C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

### Syntax of free()

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

### Example 1: malloc() and free()

// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &n);

ptr = (int\*) malloc(n \* sizeof(int));

// if memory cannot be allocated

if(ptr == NULL) {

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements: ");

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

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

// deallocating the memory

free(ptr);

return 0;

}

**Output**

Enter number of elements: 3

Enter elements: 100

20

36

Sum = 156

Here, we have dynamically allocated the memory for n number of int.

### Example 2: calloc() and free()

// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>

#include <stdlib.h>

int main() {

int n, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &n);

ptr = (int\*) calloc(n, sizeof(int));

if(ptr == NULL) {

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements: ");

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

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

free(ptr);

return 0;

}

**Output**

Enter number of elements: 3

Enter elements: 100

20

36

Sum = 156

## C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

### Syntax of realloc()

ptr = realloc(ptr, x);

Here, ptr is reallocated with a new size x.

### Example 3: realloc()

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*ptr, i , n1, n2;

printf("Enter size: ");

scanf("%d", &n1);

ptr = (int\*) malloc(n1 \* sizeof(int));

printf("Addresses of previously allocated memory:\n");

for(i = 0; i < n1; ++i)

printf("%pc\n",ptr + i);

printf("\nEnter the new size: ");

scanf("%d", &n2);

// rellocating the memory

ptr = realloc(ptr, n2 \* sizeof(int));

printf("Addresses of newly allocated memory:\n");

for(i = 0; i < n2; ++i)

printf("%pc\n", ptr + i);

free(ptr);

return 0;

}

**Output**

Enter size: 2

Addresses of previously allocated memory:

26855472

26855476

Enter the new size: 4

Addresses of newly allocated memory:

26855472

26855476

26855480

26855484