**Pointers in C – Declare, initialize and use**

Pointers are the heart of C programming. It is the most distinct feature of C, which provides power and flexibility to C. Pointers separates C from other programming languages.

C programmers make extensive use of pointers, because of their numerous benefits. Below are some advantages of pointers.

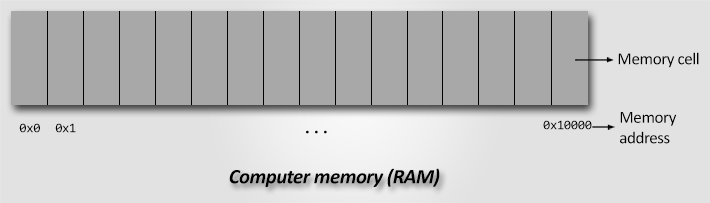
* Pointers are more efficient in handling arrays and structures.
* Pointers are used to return multiple values from a function.
* We use pointers to get reference of a variable or function.
* Pointer allows dynamic memory allocation (creation of variables at runtime) in C. Which undoubtedly is the biggest advantage of pointers.
* Pointers increases execution speed of program.

Pointers are closely related to low level memory operations. Hence, let us first understand memory in contrast to C programming.

## Understanding memory

Computer memory (RAM) is a collection of contiguous block of bytes. Where individual block is called as cell (memory cell). Each cell has a unique numeric address (also known as physical memory address) associated with it. These addresses starts from zero and runs up to maximum memory size (in bytes).

For example, memory location of a 64KB RAM starts from 0 and ends to 65536 (or 0x10000) bytes.

Computer memory of 64KB

Before I formally introduce pointers let us first see what happens during a variable definition.  
Consider the statement int num = 10;

* For the above statement, the C compiler allocates memory capable to store an integer. Let say memory is allocated at address 0x1200.
* After [memory allocation](https://codeforwin.org/2018/05/compile-time-and-runtime-memory-allocation.html), the C compiler defines a label (variable name) to access the memory location. The label is mapped to the allocated memory.
* Finally, the constant 10 is stored at 0x1200. Whenever you refer num inside your program, internally C refers to the memory location of num.

## What is a pointer?

A pointer is a variable that stores memory address. If it is a variable, it must have a [valid C data type](https://codeforwin.org/2017/08/list-data-types-c-programming.html). Yes, every pointer variable has a data type associated with it. Which means an integer pointer can hold only integer variable addresses.

***Note:****We never say pointer stores or holds a memory location. Instead, we say pointer points to a memory location. So from now always use the language pointer points to a memory location.*

## Reference operator &

Because we are dealing with memory addresses, we must know how to get

memory address of a variable. We use unary & (reference of) operator to get memory address of a variable. Reference operator is also known as **address of operator**.

### Syntax to use reference of operator

&variable-name;

### Example program to use reference operator

#include <stdio.h>

int main()

{

int num = 10;

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

/\* &num gets the address of num. \*/

printf("Address of num = %d\n", &num);

printf("Address of num in hexadecimal = %x", &num);

return 0;

}

## Dereference operator \*

Once you have a memory address, you must be willing to get value stored at that memory address, for that we need to dereference the memory address.

Dereferencing is the process of retrieving value at memory location pointed by a pointer. We use unary \* dereference operator to get value pointed by a memory address. Dereference operator is also known as **indirection operator**.

### Syntax to use dereference operator

\*memory-address-or-pointer-variable;

### Example program to use dereference operator

#include <stdio.h>

int main()

{

int num = 10;

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

/\* &num gets the address of num. \*/

printf("Address of num = %d\n", &num);

/\*

\* &num gets the address of num

\* and (\*(&num)) gets value pointed by (&num)

\*/

printf("Value of num = %d\n", \*(&num));

return 0;

}

## How to declare pointer variable

Once you got basics of memory addresses, reference and dereference operator. Let us declare our first pointer variable.

Pointer variable declaration follows almost similar syntax as of normal variable.

### Syntax to declare pointer variable

data-type \* pointer-variable-name;

* data-type is a valid [C data type](https://codeforwin.org/2017/08/data-types-in-c-programming.html).
* \* symbol specifies it is a pointer variable. You must prefix \* before variable name to declare it as a pointer.
* pointer-variable-name is a [valid C identifier](https://codeforwin.org/2017/08/keywords-identifiers-c.html#identifiers) i.e. the name of pointer variable.

### Example to declare pointer variable

int \* ptr;

In above example I declared an integer pointer.

## How to initialize pointer variable

There are two ways to initialize a pointer variable. You can use reference

operator & to get memory location of a variable or you can also directly assign one pointer variable to other pointer variable.

### Examples to initialize pointer variable

int num = 10;

int \*ptr = &num; // Assign address of num to ptr

// You can also assign a pointer variable to another

int \*ptr1 = ptr; // Initialize pointer using another pointer

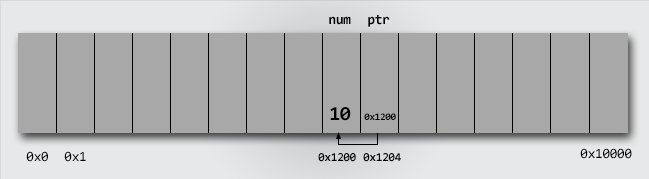
## How pointers are stored in memory

You got a basic picture of pointer working. Let us take a closer look on how pointer variables are stored in memory. Consider the following statements

int num = 10;

int \*ptr = &num;

Below is memory representation of above two statements.



**Null Pointer**

We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

Following program illustrates the use of a null pointer:

#include <stdio.h>

int main()

{

int \*p = NULL; //null pointer

printf(“The value inside variable p is:\n%x”,p);

return 0;

}

Output:

The value inside variable p is:

0

Pointer Arithmetic

Pointer is a variable that points to a memory location. Memory addresses are numeric value that ranges from zero to maximum memory size in bytes. These addresses can be manipulated like simple variables. You can increment, decrement, calculate or compare these addresses manually.

C language provides a set of operators to perform arithmetic and comparison of memory addresses. Pointer arithmetic and comparison in C is supported by following operators -

* Increment and decrement ++ and --
* Addition and Subtraction + and -
* Comparison <, >, <=, >=, ==, !=

## Pointer increment and decrement

Increment operator when used with a pointer variable returns next address pointed by the pointer. The next address returned is the sum of current pointed address and size of pointer data type.

Or in simple terms, incrementing a pointer will cause the pointer to point to a

memory location skipping N bytes from current pointed memory location. Where N is size of pointer data type.

Similarly, decrement operator returns the previous address pointed by the pointer. The returned address is the difference of current pointed address and size of pointer data type.

For example, consider the below statements.

int num = 5; // Suppose address of num = 0x1230

int \*ptr; // Pointer variable

ptr = &num; // ptr points to 0x1230 or ptr points to num

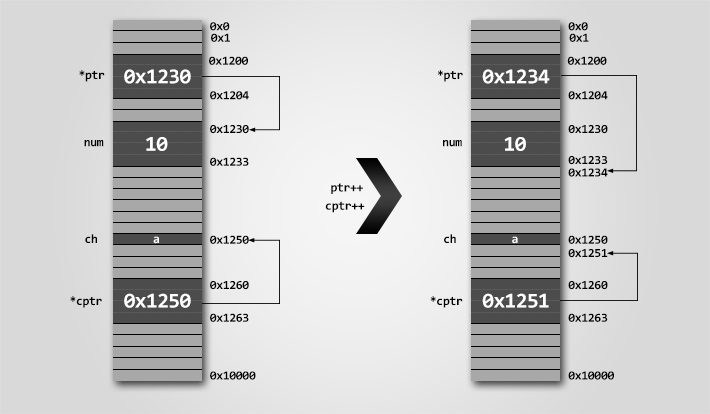
ptr++; // ptr now points to 0x1234, since integer size is 4 bytes

ptr--; // ptr now points to 0x1230

***Note:****Increment operation increments pointer address by the size of pointer data type.*

*If an integer pointer*ptr*pointing at*0x1230*, after*ptr++*it will point at*0x1234*(assuming integer size is 4 bytes).*

*If a character pointer*cptr*pointing at*0x1250*, after*cptr++*it will point at*0x1251*(since character occupies 1 byte).*



Ex: #include <stdio.h>

#define SIZE 5

int main()

{

int arr[SIZE] = {10, 20, 30, 40, 50};

int \*ptr;

int count;

ptr = &arr[0]; // ptr points to arr[0]

count = 0;

printf("Accessing array elements using pointer \n");

while(count < SIZE)

{

printf("arr[%d] = %d \n", count, \*ptr);

// Move pointer to next array element

ptr++;

count++;

}

return 0;

}

## Pointer addition and subtraction

Pointer increment operation increments pointer by one. Causing it to point to a

memory location skipping N bytes (where N is size of pointer data type).

We know that increment operation is equivalent to addition by one. Suppose an

integer pointer int \* ptr. Now, ptr++ is equivalent to ptr = ptr + 1. Similarly, you can add or subtract any integer value to a pointer.

Adding K to a pointer causes it to point to a memory location skipping K \* N bytes. Where K is a constant integer and N is size of pointer data type.

Let us revise the above program to print array using pointer.

### Example program to demonstrate pointer addition and subtraction

#include <stdio.h>

#define SIZE 5

int main()

{

int arr[SIZE] = {10, 20, 30, 40, 50};

int \*ptr;

int count;

ptr = &arr[0]; // ptr points to arr[0]

count = 0;

printf("Accessing array elements using pointer \n");

while(count < SIZE)

{

printf("arr[%d] = %d \n", count, \*(ptr + count));

count++;

}

return 0;

}

* When count = 0, (ptr + count) is equivalent to (ptr + 0) which points to arr[0] and hence prints 10.
* When count = 1, (ptr + count) is equivalent to (ptr + 1) which points to arr[1] and hence prints 20.
* Similarly when count = 4, (ptr + count) is equivalent to (ptr + 4) which points to arr[4] and hence prints 50.

## Pointer comparison

In C, you can compare two pointers using [relational operator](https://codeforwin.org/2017/08/relational-operators-c.html). You can perform six different type of pointer comparison <, >, <=, >=, == and !=.

***Note:****Pointer comparison compares two pointer addresses to which they point to, instead of comparing their values.*

Pointer comparisons are less used when compared to pointer arithmetic. However, I frequently use pointer comparison when dealing with arrays.

Pointer comparisons are useful,

* If you want to check if two pointer points to same location. For example,

int main()

{

int num = 10;

int \*ptr1 = &num; // ptr1 points to num

int \*ptr2 = &num; // ptr2 also points to num

if(ptr1 == ptr2)

{

// Both pointers points to same memory location

// Do some task

}

return 0;

}

If you want to check if a pointer points within an array range. For example,

int main()

{

int arr[5] = {10, 20, 30, 40, 50};

int \*ptr = &arr[0]; // ptr points to arr[0]

while(ptr <= &arr[4])

{

// ptr will always point within the array

// Do some task

// Move ptr to next array element

ptr++;

}

}

#include <stdio.h>

#define SIZE 5

int main()

{

int arr[SIZE] = {10, 20, 30, 40, 50};

int \*ptr = &arr[0]; // ptr points to arr[0]

printf("Accessing array elements using pointer \n");

while(ptr < &arr[SIZE])

{

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

// Move to next array element

ptr++;

}

return 0;

}

## Rules for performing pointer arithmetic

Pointer arithmetic can be a nightmare if not used correctly. Incorrect pointer arithmetic will lead to you compilation error as well as program crash.

Following are some rules that you must mind while performing pointer arithmetic.

* Result of two pointer addition or subtraction is an integer. For example,

int arr[] = {10, 20, 30, 40, 50};

int \*ptr1 = &arr[0];

int \*ptr2 = &arr[4];

int \*ptr3 = ptr2 - ptr1; // ERROR -> (ptr2 - ptr1) evaluates to integer not integer pointer

* Result of pointer and integer addition or subtraction is a pointer. For example,

int arr[] = {10, 20, 30, 40, 50};

int \*ptr = &arr[0];

ptr = (ptr + 2); // ptr will now point to arr[2]

* You must not use multiplication and division operator with pointers.

### Valid and invalid examples of pointer arithmetic

int num=10, k=2; // Integer variable

int \*ptr1, \*ptr2, \*ptr3; // Integer pointers

ptr1 = ptr1 – 2; // Valid

ptr1 = ptr1 – k; // Valid

ptr3 = ptr2 – ptr1; // Invalid, non-portable pointer conversion.

// Missing cast. See rule 1.

ptr3 = (int \*) (ptr2 – ptr1) // Valid

ptr3 = ptr2 – ptr1 - k; // Invalid, non-portable pointer conversion.

// Missing cast. See rule 1.

ptr3 = (int \*)(ptr2 – ptr1) + k; // Valid

ptr1 = ptr1 + 2; // Valid

ptr1 = ptr1 + k; // Valid

ptr3 = ptr1 + ptr2; // Invalid, non-portable pointer conversion.

// Missing cast. See rule 1.

ptr3 = (int \*) (ptr1 + ptr2) // Valid

ptr3 = ptr1 + ptr2 + k; // Invalid, non-portable pointer conversion.

// Missing cast. See rule 1.

ptr3 = (int \*)(ptr1 + ptr2) + k; // Valid

ptr1 = ptr1 \* 2; // Invalid, illegal use of pointer. See rule 3.

ptr1 = ptr1 \* k; // Invalid, illegal use of pointer. See rule 3.

ptr3 = ptr2 \* ptr1; // Invalid, illegal use of pointer. See rule 3.

ptr1 = ptr1 / 2; // Invalid, illegal use of pointer. See rule 3.

ptr1 = ptr1 / k; // Invalid, illegal use of pointer. See rule 3.

* ptr3 = ptr2 / ptr1; // Invalid, illegal use of pointer. See rule 3.

**Advantages of Pointers in C**

* Pointers are useful for accessing memory locations.
* Pointers provide an efficient way for accessing the elements of an array structure.
* Pointers are used for dynamic memory allocation as well as deallocation.
* Pointers are used to form complex data structures such as linked list, graph, tree, etc.

**Disadvantages of Pointers in C**

* Pointers are a little complex to understand.
* Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.
* If an incorrect value is provided to a pointer, it may cause memory corruption.
* Pointers are also responsible for memory leakage.
* Pointers are comparatively slower than that of the variables.
* Programmers find it very difficult to work with the pointers; therefore it is programmer's responsibility to manipulate a pointer carefully.

Ex : Swapping

void swap (int \*a, int \*b);

int main()

{

int m = 25;

int n = 100;

printf("m is %d, n is %d\n", m, n);

swap(&m, &n);

printf("m is %d, n is %d\n", m, n);

return 0;

}

void swap (int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

Output:

* m is 25, n is 100
* m is 100, n is 25

Ex: Simple Calculator

#include<stdio.h>

void cal(float a,float b,float \*sum,float \*diff,float \*pro,float \*div);

main()

{

float a,b,s,d,p,q;

printf("Enter a and b\n");

scanf("%f%f",&a,&b);

cal(a,b,&s,&d,&p,&q);

printf("\n sum of two numbers=%f\n",s);

printf("\n diff of two numbers=%f\n",d);

printf("\n product of two numbers=%f\n",p);

printf("\n quotient of two numbers=%f\n",q);

}

void cal(float a,float b,float \*sum,float \*diff,float \*pro,float \*div)

{

\*sum=a+b;

\*diff=a-b;

\*pro=a\*b;

\*div=a/b;

}

\*

\* C program to find the sum of all elements of an array using

\* pointers as arguments.

\*/

#include <stdio.h>

int addnum(int \*ptr,int);

void read(int \*array,int size);

void main()

{

int array[10],n;

int sum;

scanf("%d",&n);

read(array,n);

sum = addnum(array,n);

printf("Sum of all array elements = %5d\n", sum);

}

void read(int \*array,int size)

{int i;

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

{ scanf("%d", (array + i));

}}

int addnum(int \*ptr,int n)

{

int index, total = 0;

for (index = 0; index < n; index++)

{

total += \*(ptr + index);

}

return(total);

}

// C program to find max and min element from the list of n numbers

#include <stdio.h>

#define MAX\_SIZE 100

/\* Function declaration \*/

void getMinMax(int \* numbers, int size, int \* min, int \* max);

void read(int \*numbers,int size);

int main()

{

int arr[MAX\_SIZE], size;

int min, max;

printf("Enter size of array: ");

scanf("%d", &size);

printf("Enter %d elements in array: ", size);

read(arr,size);

// Call min max function to get minimum and maximum value.

getMinMax(arr, size, &min, &max);

printf("Minimum element in array: %d\n", min);

printf("Maximum element in array: %d\n", max);

system("pause");

return 0;

}

void read(int \*numbers,int size)

{int i;

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

{ scanf("%d", (numbers + i));

}

//

}

void getMinMax(int \* numbers, int size, int \* min, int \* max)

{

int i;

\*min = \*(numbers + 0);

\*max = \*(numbers + 0);

for (i = 1; i < size; i++)

{

// Check minimum number

if(\*(numbers + i) < \*(min))

\*min = \*(numbers + i);

// Check maximum number

if(\*(numbers + i) > \*(max))

\*max = \*(numbers + i);

}

}