Acessing memory allocations of the symbols

Organised & Supported by **RuggedBOARD**

Agenda

- Pointers
- How to use pointers?
- NULL pointers
- Pointers Arithmetic,Increment,decrement,Comparition
- Arrays of pointers
- Array pointers
- Dynamic Memory Allocation
- Pointer to pointer

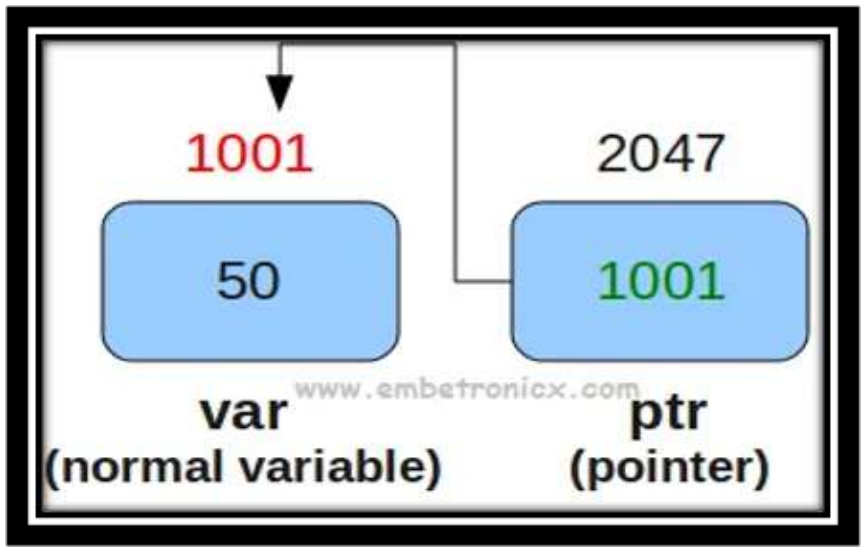
Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.

Pointers are special variables that can hold the address of a variable.

Since they store the memory address of a variable, the pointers are very commonly said to “point to variables”.

A normal variable **‘var’** has a memory address of 1001 and holds a value of 50.
A pointer variable has its own address 2047 but stores 1001, which is the address of the variable ‘**var**’.



Pointers

How to Declare a Pointer?

<pointer type> *<pointer-name>;

Pointer-type:

It specifies the type of pointer.
It can be int, char, float, etc

Pointer-name:

It can be any name specified by the user.

Ex: char *chptr;
'char' signifies the pointer type,
chptr is the name of the pointer
'*' signifies that **'chptr'** is a pointer variable.

How to initialize a Pointer?

<pointer declaration(except semicolon)> = <address of a variable>

(OR)

<pointer declaration> <name-of-pointer> = <address of a variable>

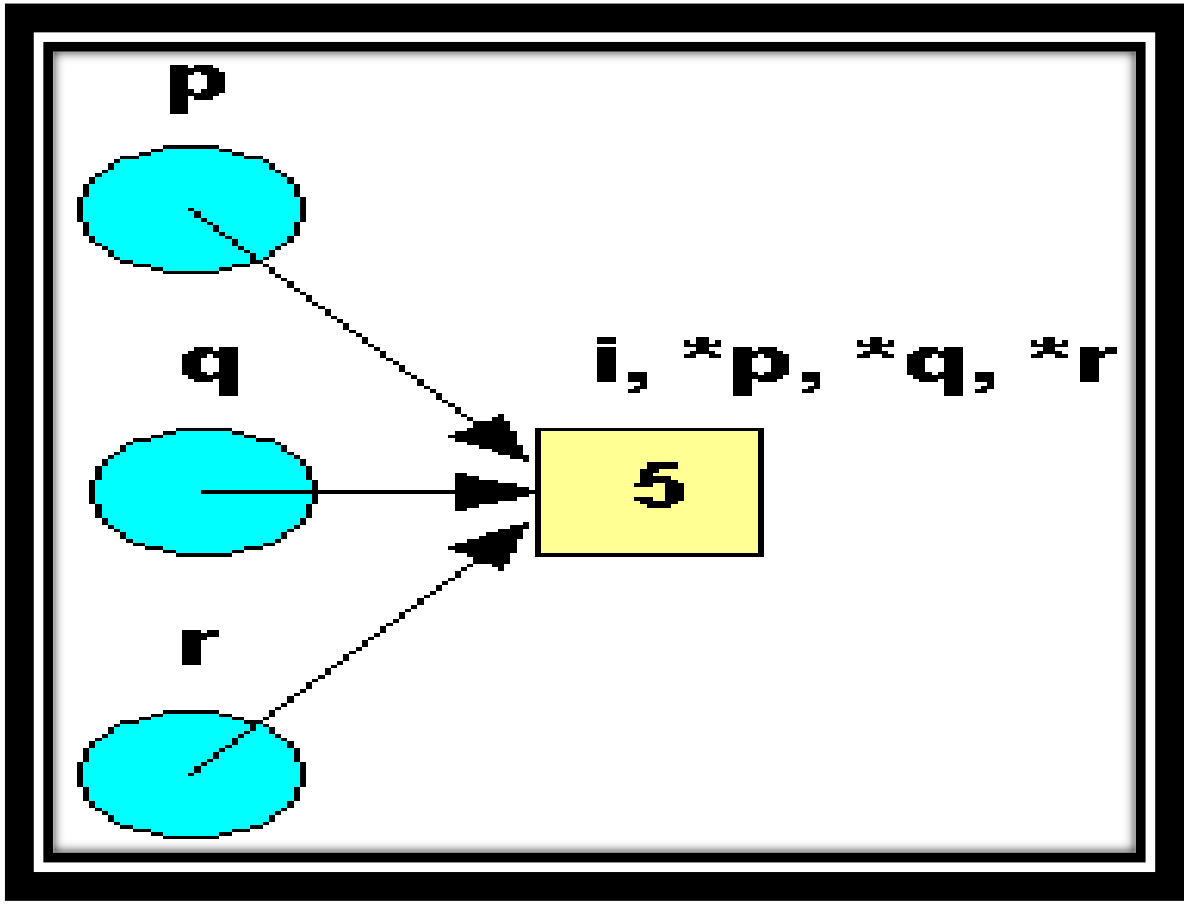
Ex :

```
char  ch = 'c';
char  *chptr = &ch; //initialize
```

How to Use a Pointer?

Any number of pointers can point to the same address.
Eg- Declare p, q, and r as integer pointers
set all of them to point to **i**.

```
#include<stdio.h>
int main()
{
    int I = 5;
    int *p, *q, *r;
    p = &i;
    q = &i;
    r = p;
    Printf("%d , %d,%d,%d",I,*p,*q,*r);
    return 0;
}
```

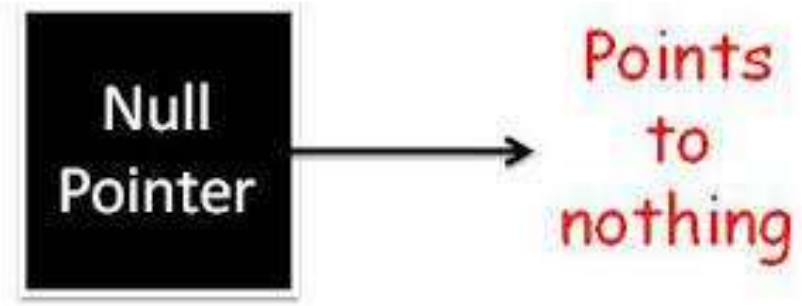


After executing this code, the variable i has four names:
i, *p, *q and *r

NULL Pointers

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned,done at the time of variable declaration.
- A pointer that is assigned NULL is called a null pointer.
- NULL pointer is a constant with a value of zero defined in several standard libraries.

```
#include <stdio.h>
int main ()
{
        int *ptr = NULL;
        printf("The value of ptr is : %p\n", &ptr);
        return 0;
}
```



NOTE:gives segmentation fault core dumped for int *ptr = NULL, when we print value

Pointer arithmetic

- C pointer is an address, which is a numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and -.
- Let us consider that ptr is an integer pointer which points to the address 1000.
- Ex : ptr++

  Now, after the above operation, the ptr will point to the location 1004.
- If ptr points to a character whose address is 1000.
- Ex : ptr++

  Then above operation will point to the location 1001.

- Variable pointer can be incremented.
- variable pointer to access each succeeding element of the array:

```
int var[] = { 10 , 100 , 200 };
int i, *ptr;
/* let us have array address in pointer */
 ptr = var;
 /* move to the next location */
 ptr++;
```

Decrementing a pointer, which decreases its value by the number of bytes of its data type.

```
int var[] = { 10 , 100 , 200 };
int i, *ptr;
ptr = &var[MAX- 1 ];//Here MAX is 3
/* move to the previous location */
ptr--;
```

Pointer arithmetic

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

    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

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

    /* move to the previous location */
    ptr--;
  }
  return 0;
}
```

Pointer arithmetic

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.

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

    /* point to the next location */
    ptr++;
    i++;
  }
  return 0;
}
```

Array of pointers

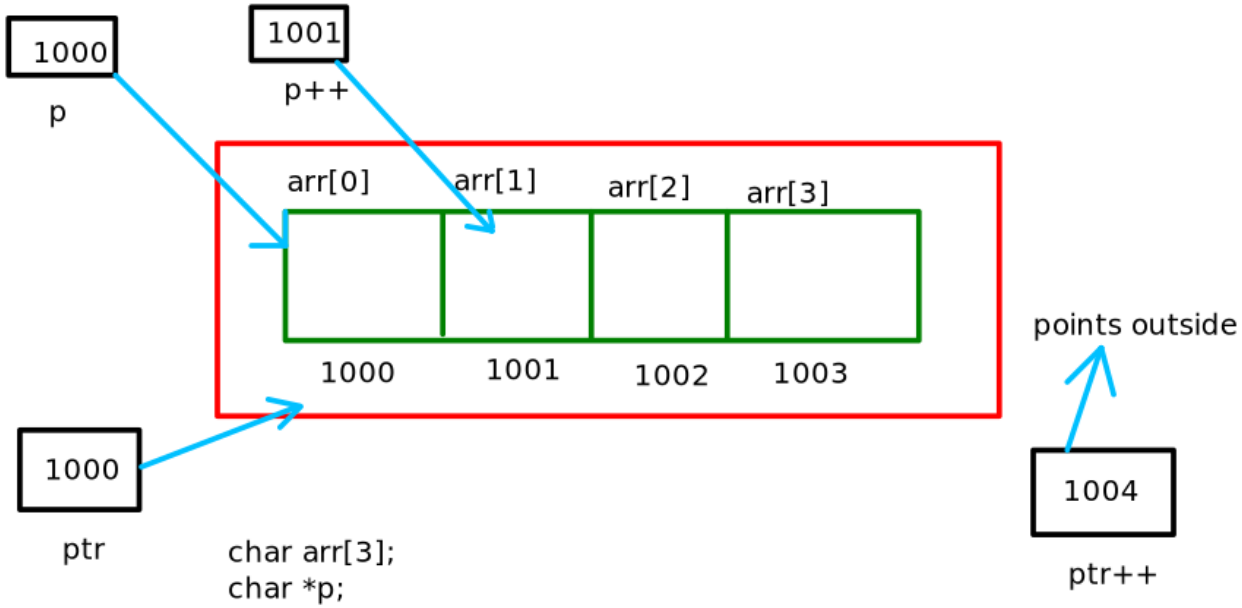
What is an Array of Pointers in C?
When we want to point at multiple variables or memories of the same data type in a C program, we use an array of pointers.
Let us assume that a total of five employees are working at a cheesecake factory. We can store the employees' names in the form of an array. Along with this, we also store the names of employees working at the office, the library, and the shoe store.

```
#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;
}
```

```
#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;
}
```

Array pointer or pointer to an array

```
// C program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include<stdio.h>
int main()
{
   // Pointer to an integer
   int *p;
    // Pointer to an array of 5 integers
   int (*ptr)[5];
   int arr[5];
   // Points to 0th element of the arr.
   p = arr;
   // Points to the whole array arr.
   ptr = &arr;
   printf("p = %p, ptr = %p\n", p, ptr);
   p++;
   ptr++;
   printf("p = %p, ptr = %p\n", p, ptr);
   return 0;
}
```



Dynamic Memory Allocation

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

malloc()

calloc()

realloc()

free()

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

Difference between static memory allocation and dynamic memory allocation

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Malloc()

- The **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size.
- It returns a pointer of type void which can be cast into a pointer of any form.
- It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax:

ptr=(cast-type*)malloc(byte-size)

Malloc()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int* ptr;
	int n, i;
	printf("Enter number of elements:");
	scanf("%d",&n);
	printf("Entered number of elements: %d\n", n);
	ptr = (int*)malloc(n * sizeof(int));
	if (ptr == NULL)
	{
		printf("Memory not allocated.\n");
		exit(0);
	}
```

```
else
	{
		// Memory has been successfully allocated
		printf("Memory successfully allocated using malloc.\n");

		// Get the elements of the array
		for (i = 0; i < n; ++i)
		{
			ptr[i] = i + 1;
		}

		// Print the elements of the array
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i)
		{
			printf("%d, ", ptr[i]);
		}
		printf("\n");
	}
	return 0;
}
```

Calloc()

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

1.It initializes each block with a default value ‘0’.

2.It has two parameters or arguments as compare to malloc().

Syntax:

ptr = (cast-type*)calloc(n, element-size);

here, n is the number of elements and element-size is the size of each element.

Calloc()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int* ptr;
	int n, i;

	// Get the number of elements for the array
	n = 5;
	// Dynamically allocate memory using calloc()
	ptr = (int*)calloc(n, sizeof(int));

	// Check if the memory has been successfully
	// allocated by calloc or not
	if (ptr == NULL)
	{
		printf("Memory not allocated.\n");
		exit(0);
	}
```

```
else
{
		// Memory has been successfully allocated
		printf("Memory successfully allocated using calloc.\n");

		// Get the elements of the array
		for (i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}
		// Print the elements of the array
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i) {
			printf("%d, ", ptr[i]);
		}
		printf("\n");
	}
	return 0;
}
```

free()

“free” method in C is used to dynamically **de-allocate** the memory.
The memory allocated using functions malloc() and calloc() is not de-allocated on their own.
Hence the free() method is used, whenever the dynamic memory allocation takes place.
It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```

free()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int* ptr;
	int n, i;

	// Get the number of elements for the array
	n = 5;
	// Dynamically allocate memory using calloc()
	ptr = (int*)calloc(n, sizeof(int));

	// Check if the memory has been successfully
	// allocated by calloc or not
	if (ptr == NULL)
	{
		printf("Memory not allocated.\n");
		exit(0);
	}
```

```
else
{
		// Memory has been successfully allocated
		printf("Memory successfully allocated using calloc.\n");

		// Get the elements of the array
		for (i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}
		// Print the elements of the array
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i) {
			printf("%d, ", ptr[i]);
		}
		printf("\n");
	}
	free(ptr);
	ptr = NULL;
	return 0;
}
```

realloc()

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory.
In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.
Re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax:
ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.

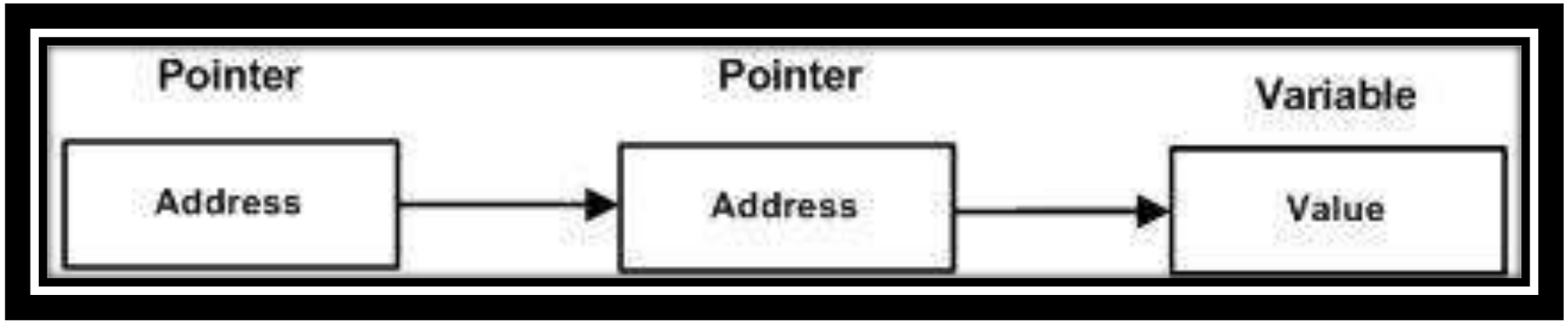
realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int* ptr;
	int n, i;
	n = 5;
	printf("Enter number of elements: %d\n", n);
	ptr = (int*)calloc(n, sizeof(int));

	 if (ptr == NULL) {
		printf("Memory not allocated.\n");
		exit(0);
	}
	else {
printf("Memory successfully allocated using calloc.\n");
		for (i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}
```

```
  printf("The elements of the array are: ");
  for (i = 0; i < n; ++i) {
		printf("%d, ", ptr[i]);
   }
   n = 10;
  printf("\n\nEnter the new size of the array: %d\n", n);
  ptr = realloc(ptr, n * sizeof(int));
 printf("Memory successfully re-allocated using realloc.\n");
    for (i = 5; i < n; ++i) {
		ptr[i] = i + 1;
	}
	printf("The elements of the array are: ");
	for (i = 0; i < n; ++i) {
		printf("%d, ", ptr[i]);
	}
	free(ptr);
	}
	return 0;
}
```

Pointer to Pointer

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.
- first pointer contains the address of the second pointer.
- which points to the location that contains the actual value




```
int **var;
```

Pointer to Pointer

```
#include <stdio.h>
// C program to demonstrate pointer to pointer
int main()
{
	int var = 789;
	// pointer for var
	int *ptr2;
	// double pointer for ptr2
	int **ptr1;
	// storing address of var in ptr2
	ptr2 = &var;
	// Storing address of ptr2 in ptr1
	ptr1 = &ptr2;
	// Displaying value of var using
	// both single and double pointers
	printf("Value of var = %d\n", var );
	printf("Value of var using single pointer = %d\n", *ptr2 );
	printf("Value of var using double pointer = %d\n", **ptr1);

	return 0;
}
```

Thank You