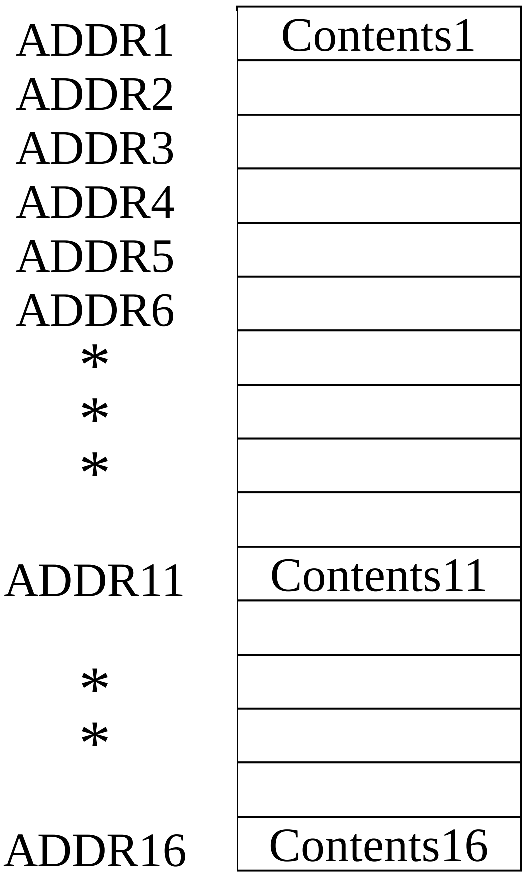
Pointers

POINTERS

- Pointers are variables that contain ***memory addresses*** as their values.
- A variable name ***directly*** references a value.
- A pointer ***indirectly*** references a value. Referencing a value through a pointer is called ***indirection***.
- A pointer variable must be declared before it can be used.

Concept of Address and Pointers

- **Memory can be conceptualized as a linear set of data locations.**
- **Variables reference the contents of a locations**
- **Pointers have a value of the address of a given location**



POINTERS

- **Examples of pointer declarations:**

  int *a**;**

  float *b**;**

  char *c**;**

- **The asterisk, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data that the pointer points to, but NOT the name of the variable pointed to.**

POINTERS: Example

```
#include <stdio.h>
int main ( )
{
int *aptr ;                /* Declare a pointer to an int */
float *bptr ;              /* Declare a pointer to a float */
int a =5 ;                 /* Declare an int variable */
float b = 5.5;             /* Declare a float variable */
aptr = &a;
bptr = &b;
printf("Apointer holds %d and B pointer holds %d",
    aptr,bptr);
}
```

Use of & and *

- **When is & used?**

- **When is * used?**

- **& -- "address operator or referencing operator" which gives or produces the memory address of a data variable**
- *** -- "dereferencing operator" which provides the contents in the memory location specified by a pointer**

Pointers and Functions

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.

- We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap.

- This is known as "call by value".

Pointers and Functions

- If instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference" since we are ***referencing*** the variables.

- The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now actually swapped when the control is returned to main function.

Arithmetic and Logical Operations on Pointers

- **A pointer may be incremented or decremented**
- **An integer may be added to or subtracted from a pointer.**
- **Pointer variables may be subtracted from one another.**
- **Pointer variables can be used in comparisons, but usually only in a comparison to NULL.**

NULL pointer

- A pointer pointing to nothing or no memory location is called null pointer. For doing this we simply assign NULL to the pointer.
- So while declaring a pointer we can simply assign NULL to it in following way.

  int *p = **NULL**;

- NULL is a constant which is already defined in C and its value is 0. So instead of assigning NULL to pointer while declaring it we can also assign 0 to it.

Usage of Null Pointer

1. We can simply check the pointer is NULL or not before accessing it. This will prevent crashing of program or undesired output.

```
int *p = NULL;
if(p != NULL)
 {
//here you can access the pointer
}
```

Usage of Null Pointer

2. A pointer pointing to a memory location even after its de-allocation is called dangling pointer. When we try to access dangling pointer it crashes the program. So to solve this problem we can simply assign NULL to it.

```
void function()
{
    int *ptr = (int *)malloc(SIZE);
    . . . . . .
    free(ptr);    //ptr now becomes dangling pointer which is pointing to dangling reference
    ptr=NULL;    //now ptr is not dangling pointer
}
```

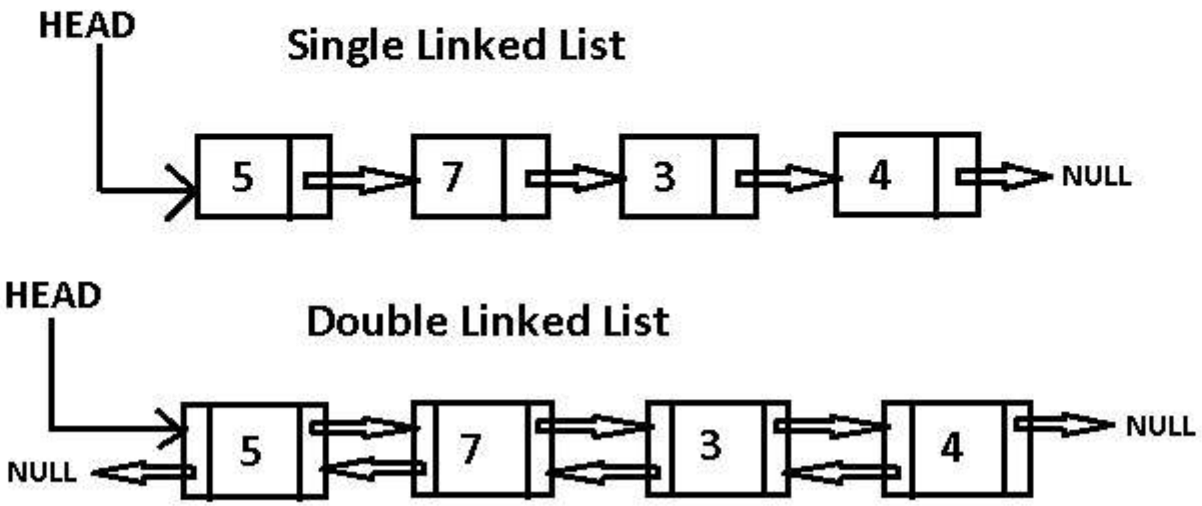
Usage of Null Pointer

3. We can simply pass NULL to a function if we don't want to pass a valid memory location.

```
void fun(int *p)
 {
   //some code
}
int main()
{
   fun(NULL);
   return 0;
}
```

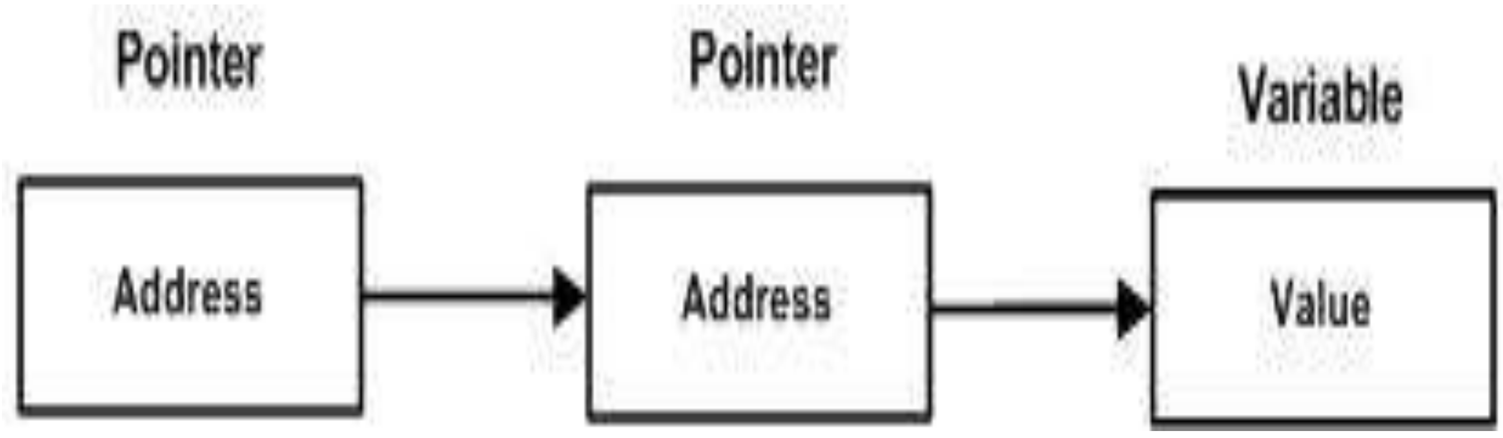
Usage of Null Pointer

4. Null pointer is also used to represent the end of a linked list.



Chain of Pointers

- 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



Chain of Pointers

- 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.

  int **var;

Example

```
#include <stdio.h>
 int main ()
{
int var;
int *ptr;
 int **pptr;
 var = 3000;
ptr = &var;
pptr = &ptr;
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;
}
```

Arrays and Pointers

- **We declare an array using [ ] in our declaration following the variable name**

```
int x[5];
```

- **We can interact with the array elements either through pointers or by using [ ].**

- **When an array name is used by itself, the array's address (First Element) is returned. We can assign this address to a pointer as illustrated below:**

```
int vector[5] = {1, 2, 3, 4, 5};
int *pv = vector;
```

Arrays and Pointers

- The variable pv is a pointer to the first element of the array and not the array itself.
- We can use either the array name by itself or use the address-of operator with the array’s first element as illustrated below. These are equivalent and will return the address of vector

```
printf("%p\n",vector);
printf("%p\n",&vector[0]);
```

Arrays and Pointers

We can also use array subscripts with pointers. Effectively, the notation pv[i] is evaluated as:

***(pv + i)**

The following three statements are equivalent:

***(pv + i)**

***(vector + i)**

vector[i];

Arrays and Pointers

The following Three statements are equivalent:

```
(pv + i);
(vector + i);
&vector[i];
```

Example

- #include<stdio.h>
- int main()
- {
- char vowels[] = {'A', 'E', 'I', 'O', 'U'};
- char *pvowels;
- pvowels=vowels;
- int i;
- for (i = 0; i < 5; i++) {
- printf("&vowels[%d]: %u, pvowels + %d: %u, vowels + %d: %u\n", i, &vowels[i], i, pvowels + i, i, vowels + i);
- }
- for (i = 0; i < 5; i++) {
- printf("vowels[%d]: %c, *(pvowels + %d): %c, *(vowels + %d): %c\n", i, vowels[i], i, *(pvowels + i), i, *(vowels + i));
- }}

Example2

```
#include<stdio.h>
int main()
{
    int *p; /*Pointer to an integer*/
    int (*ptr)[5]; /* Pointer to an array of 5 integers*/
    int arr[5]={1,2,3,4,5};
    p = arr; /*Points to 0th element of the arr*/
    ptr = &arr;
    printf("p = %d, ptr = %d\n", p, ptr);
    p++; /*Points to the whole array arr*/
    ptr++;
    printf("p = %d, ptr = %d\n", p, ptr);
     printf("%d\n",sizeof(*ptr));
     printf("%d",sizeof(p));       return 0;
}
```

Pointer to Multidimensional Array

- A multidimensional array is of form, **a[i][j]**.
- As is 1D array, name of the array gives its base address. In **a[i][j]**, **a** will give the base address of this array, even **a + 0 + 0** will also give the base address, that is the address of **a[0][0]**element.

***(*(a + i) + j)** which is same as, **a[i][j]**

For 3 multidimensional array of form, **a[i][j][k]**.

***(*(*(a + i) + j) +k)**

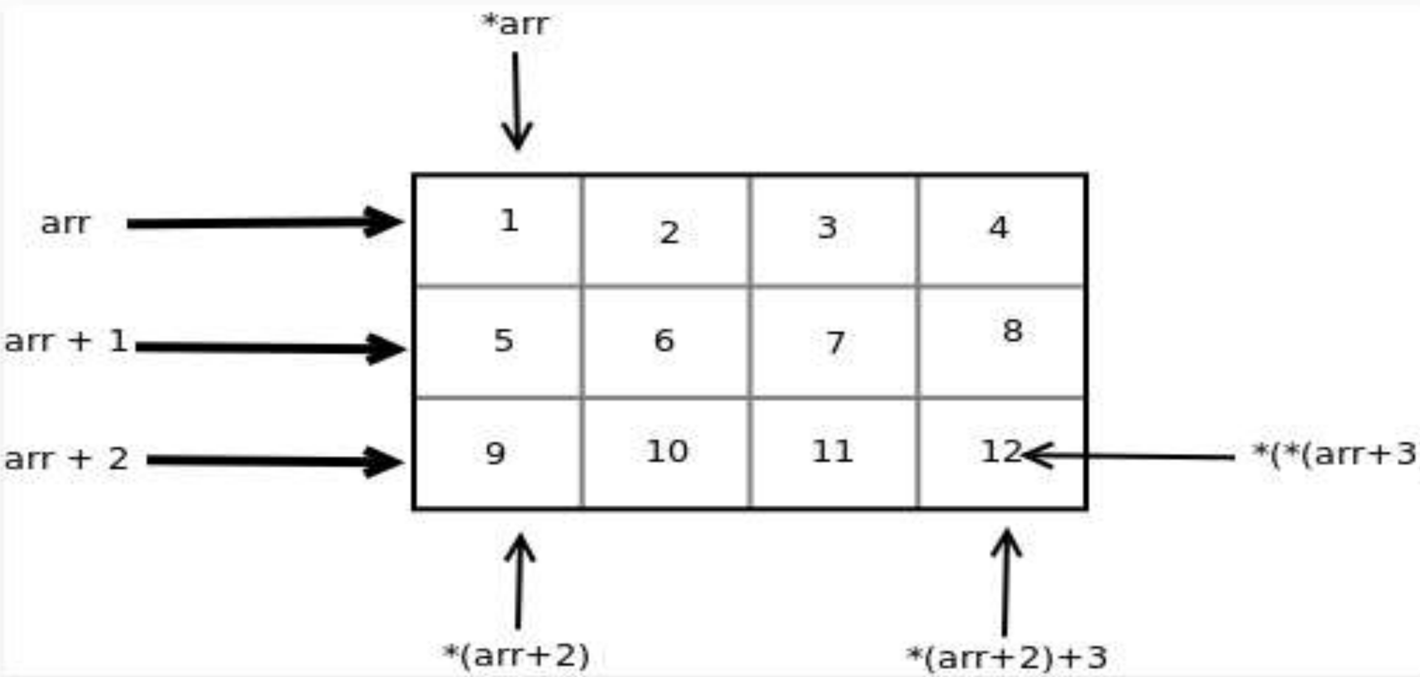
Example

```
#include<stdio.h>
int main()
{
  int arr[3][4] = {   { 10, 11, 12, 13 }, { 20, 21, 22, 23 }, { 30, 31, 32, 33 } };
  int i, j;
  for (i = 0; i < 3; i++)
  {
    printf("Address of %dth array = %p %p\n",   i, arr[i], *(arr + i));
    for (j = 0; j < 4; j++)
      printf("%d %d ", arr[i][j], *(*(arr + i) + j));
    printf("\n");
  }
  return 0;
}
```

Example:Continue..

arr	-	Points to 0^{th} element of arr	-	Points to 0^{th} 1-D array
arr + 1	-	Points to 1^{th} element of arr	-	Points to 1^{nd} 1-D array
arr + 2	-	Points to 2^{th} element of arr	-	Points to 2^{nd} 1-D array

arr	**Points to 0^{th} 1-D array**
***arr**	**Points to 0^{th} element of 0^{th} 1-D array**
(arr + i)	**Points to i^{th} 1-D array**
***(arr + i)**	**Points to 0^{th} element of i^{th} 1-D array**
***(arr + i) + j)**	**Points to j^{th} element of i^{th} 1-D array**
***(*(arr + i) + j)**	**Reprents the value of j^{th} element of i^{th} 1-D array**



Pointer: Dynamic Memory Allocation

- An array is a collection of fixed number of values of a single type. That is, you need to declare the size of an array before you can use it.
- Sometimes, the size of 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.
- Four library functions under <stdlib.h> makes dynamic memory allocation in C programming. They are malloc(), calloc(), realloc() and free().

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 of type void which can be casted into pointer of any form.

Syntax

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

Example:

ptr = (int*) malloc(100 * sizeof(int));

calloc()

- The name "calloc" stands for contiguous allocation.
- The malloc() function allocates a single block of memory. Whereas, calloc() allocates multiple blocks of memory and initializes them to zero.

  Syntax of calloc()

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

  Example:

  ptr = (float*) calloc(25, sizeof(float));
- This statement allocates contiguous space in memory for 25 elements each with the size of float.

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:malloc() and free()

```
#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(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;
 }
```

Example: calloc() and free()

```
#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;
 }
```

realloc()

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

Syntax of realloc()

ptr = realloc(ptr, x);

Here, ptr is reallocated with new size x.

Example:realloc()

```
#include <stdio.h>
 #include <stdlib.h>
 int main()
{ int *ptr, i , n1, n2;
printf("Enter size of array: ");
scanf("%d", &n1);
ptr = (int*) malloc(n1 * sizeof(int));
printf("Addresses of previously allocated memory: ");
for(i = 0; i < n1; ++i) {
printf("%u\n",ptr + i); }
printf("\nEnter new size of array: ");
scanf("%d", &n2);
ptr = realloc(ptr, n2 * sizeof(int));
 printf("Addresses of newly allocated memory: ");
 for(i = 0; i < n2; ++i) printf("%u\n", ptr + i); return 0; }
```

Pointer Example: Return multiple Values 1

```
#include <stdio.h>
#include<stdlib.h>
int* initialize()
{
    int *temp = (int*) malloc(sizeof(int) * 3);
    *temp = 10; *(temp + 1) = 20; *(temp + 2) = 30;
    return temp;
}
int main(void)
{
    int a, b, c; int *arr = initialize();
    a = arr[0]; b = arr[1]; c = arr[2];
    printf("a = %d, b = %d, c = %d", a, b, c);
    return 0;
}
```

Pointer Example: Return multiple Values 2

```
#include <stdio.h>
#include<stdlib.h>
int* initialize()
{
     int arr[3]={10,20,30};
    int *temp = arr;
    return temp;
}
int main(void)
{
    int a, b, c; int *arr = initialize();
    a = arr[0]; b = arr[1]; c = arr[2];
    printf("a = %d, b = %d, c = %d", a, b, c);
    return 0;
}
```