- One other pointer conversion is allowed: **You can convert an integer into a pointer or a pointer into an integer.** However, you must use an explicit cast, and the result of such conversion is implementation defined and may result in undefined behavior. (A cast is not needed when converting zero, which is the null pointer.)
    - int *NumRecPrinted = NULL;
    - int no_of_records = 10;
    - NumRecPrinted = (int*)no_of_records;

8.5.3 Pointer Arithmetic

- There are only two arithmetic operations that you can use on pointers: addition and subtraction.
- let **p1** be an integer pointer with a current value of 2000. Also, assume **ints** are 4 bytes long. After the expression

```
p1++;
```

  p1 contains 2004, not 2001.
  The reason for this is that each time **p1** is incremented, it will point to the next integer.
- The same is true of decrements. For example, assuming that **p1** has the value 2000, the expression

```
p1--;
```

  causes **p1** to have the value 1996.

- All pointers will increase or decrease by the length of the data type they point to. This approach ensures that a pointer is always pointing to an appropriate element of its base type.

- You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

```
p1 = p1 + 12;
```

  makes **p1** point to the 12th element of **p1**'s type beyond the one it currently points to.

- Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You can subtract one pointer from another in order to find the number of objects of their base type that separate the two.

```
i = &arr[1] ;
j = &arr[5] ;
printf ( "%d", j - i, ) ; // will print 4 not 16
```

- **All other arithmetic operations are prohibited.**
- Specifically, you cannot multiply or divide pointers; you cannot add two pointers; you cannot apply the bitwise operators to them; and you cannot add or subtract type **float** or **double** to or from pointers.

```
        register int t;
        for(t=0; s[t]; ++t)
            putchar(s[t]);
    }
    /* Access s as a pointer. */
    void putstr(char *s)
    {
        while(*s)
            putchar(*s++);
    }
```

8.7 Pointers and 2D Arrays

```
/* Demo: 2-D array is an array of arrays */
void main( )
{
    int s[4][2] = {
            { 1234, 56 },
            { 1212, 33 },
            { 1434, 80 },
            { 1312, 78 }
    } ;
    int i ;
    for ( i = 0 ; i <= 3 ; i++ )
            printf ( "\nAddress of %d th 1-D array = %u", i, s[i] ) ;
}
```

OUTPUT

```
Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65516
Address of 2 th 1-D array = 65524
Address of 3 th 1-D array = 65532
```


The compiler knows that **s** is an array containing 4 one-dimensional arrays, each containing 2 integers. Thus, the expressions **s[0]** and **s[1]** would yield the addresses of the zeroth and first one-dimensional array respectively.

Suppose we want to refer to the element **s[2][1]** using pointers. We know that **s[2]** would give the address 65524. Obviously (65524 + 1) Or **(s[2] + 1)** would give the address 65528. And the value at this address can be obtained by using the value at address operator, saying ***(s[2] + 1)**.
We have already studied while learning one-dimensional arrays that **num[i]** is same as ***(num + i)**. Similarly, ***(s[2] + 1)** is same as, ***(*(s + 2) + 1)**. Thus, all the following expressions refer to the same element,

```
s[2][1]
* ( s[2] + 1 )
* ( * ( s + 2 ) + 1 )
```

Using these concepts the following program prints out each element of a two-dimensional array using pointer notation.

```
/* Pointer notation to access 2-D array elements */
main( )
```

i.e. 4001. This address is then assigned to **p**, an **int** pointer, and then using this pointer all elements of the zeroth 1- D array are accessed. Next time through the loop when **i** takes a value 1, the expression **q + i** fetches the address of the first 1-D array. This is because, **q** is a pointer to zeroth 1-D array and adding 1 to it would give us the address of the next 1-D array. This address is once again assigned to **p**, and using it all elements of the next 1-D array are accessed.

In the third function **print( )**, the declaration of **q** looks like this:

int q[ ][4] ;

This is same as **int (*q)[4]**, where **q** is pointer to an array of 4 integers. The only advantage is that we can now use the more familiar expression **q[i][j]** to access array elements. We could have used the same expression in **show( )** as well

8.8 Arrays of Pointers

- The declaration for an **int** pointer array of size 10 is

```
int *x[10];
```

- To assign the address of an integer variable called **var** to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of **var**, write

```
*x[2]
```

- If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays: Simply call the function with the array name without any subscripts. For example, a function that can receive array **x** looks like this:

```
void display_array(int *q[])
{
    int t;
    for(t=0; t<10; t++)
      printf(''%d ", *q[t]);
}
```


- Remember, **q** is not a pointer to integers, but rather a pointer to an array of pointers to integers. Therefore you need to declare the parameter **q** as an array of integer pointers, as just shown. You cannot declare **q** simply as an integer pointer because that is not what it is. Pointer arrays are often used to hold pointers to strings. For example, you can create a function that outputs an error message given its index, as shown here:

```
void syntax_error(int num)
{
     static char *err[] = {
          "Cannot Open File\n",
          ''Read Error\n",
          "Write Error\n",
          "Media Failure\n"
     };
     printf("%s", err[num]);
}
```