Chapter 5 Pointer and Array

Ground Rules

- Switch off your handphone and pager
- Switch off your laptop computer and keep it
- No talking while lecture is going on
- No gossiping while the lecture is going on
- Raise your hand if you have question to ask
- Be on time for lecture
- Be on time to come back from the recess break to continue the lecture
- Bring your lecturenotes to lecture

5.1 Pointer Arithmetic

- pointer may point to array or non-array elements
- but pointer arithmetic must be done on same array

e.g., `int a[4], *p;`

	Address		
p	7812:	a[0]	3
	7816:	a[1]	9
	7820:	a[2]	23
	7824:	a[3]	17

array a

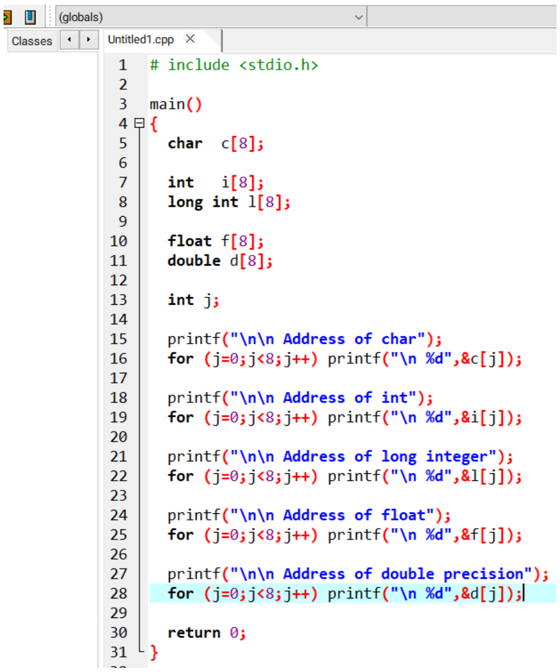
`p=a;` will direct pointer `p` to the first entry of array `a`

`*p` gives `3` (content where pointer `p` is pointing)

`*(p+2)` gives `23`

`p+3` refers to the address of `a[3]`, i.e., `7818`

What is `p+4` ?

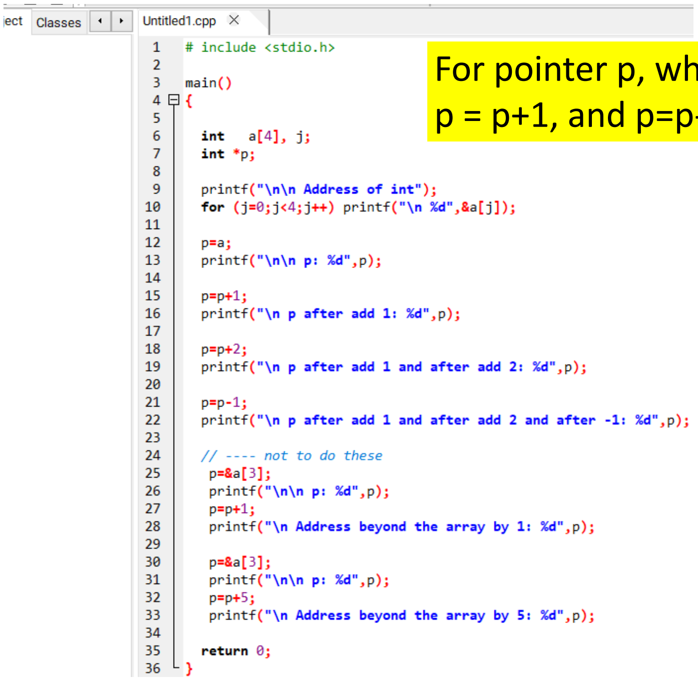


```c
# include <stdio.h>

main()
{
  char  c[8];

  int   i[8];
  long int l[8];

  float f[8];
  double d[8];

  int j;

  printf("\n\n Address of char");
  for (j=0;j<8;j++) printf("\n %d",&c[j]);

  printf("\n\n Address of int");
  for (j=0;j<8;j++) printf("\n %d",&i[j]);

  printf("\n\n Address of long integer");
  for (j=0;j<8;j++) printf("\n %d",&l[j]);

  printf("\n\n Address of float");
  for (j=0;j<8;j++) printf("\n %d",&f[j]);

  printf("\n\n Address of double precision");
  for (j=0;j<8;j++) printf("\n %d",&d[j]);

  return 0;
}
```

```
Address of char
6684180
6684181  1 byte
6684182
6684183
6684184
6684185
6684186
6684187

Address of int
6684144  4 bytes
6684148
6684152
6684156
6684160
6684164
6684168
6684172

Address of long integer
6684112
6684116  4 bytes
6684120
6684124
6684128
6684132
6684136
6684140

Address of float
6684080  4 bytes
6684084
6684088
6684092
6684096
6684100
6684104
6684108

Address of double precision
6684016
6684024  8 bytes
6684032
6684040
6684048
6684056
6684064
6684072
--------------------------------
Process exited after 11.56 seconds with return value 0
Press any key to continue . . .
```

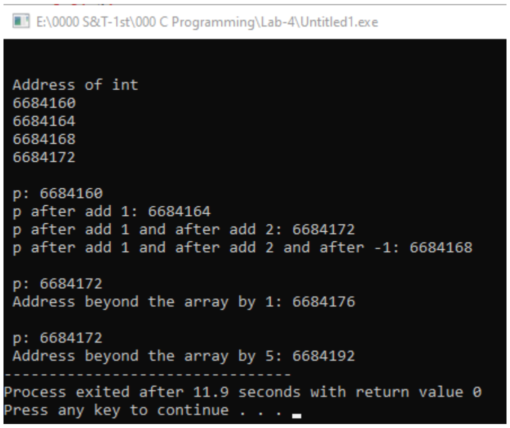
```c
# include <stdio.h>

main()
{
  char  c[8];

  int   i[8];
  long int l[8];

  float f[8];
  double d[8];

  int j;

  printf("\n\n Address of char");
  for (j=0;j<8;j++) printf("\n %d",&c[j]);

  printf("\n\n Address of int");
  for (j=0;j<8;j++) printf("\n %d",&i[j]);

  printf("\n\n Address of long integer");
  for (j=0;j<8;j++) printf("\n %d",&l[j]);

  printf("\n\n Address of float");
  for (j=0;j<8;j++) printf("\n %d",&f[j]);

  printf("\n\n Address of double precision");
  for (j=0;j<8;j++) printf("\n %d",&d[j]);

  return 0;
}
```

Source Code



```
# include <stdio.h>

main()
{

   int   a[4], j;
   int *p;

   printf("\n\n Address of int");
   for (j=0;j<4;j++) printf("\n %d",&a[j]);

   p=a;
   printf("\n\n p: %d",p);

   p=p+1;
   printf("\n p after add 1: %d",p);

   p=p+2;
   printf("\n p after add 1 and after add 2: %d",p);

   p=p-1;
   printf("\n p after add 1 and after add 2 and after -1: %d",p);

   // ---- not to do these
    p=&a[3];
    printf("\n\n p: %d",p);
    p=p+1;
    printf("\n Address beyond the array by 1: %d",p);

    p=&a[3];
    printf("\n\n p: %d",p);
    p=p+5;
    printf("\n Address beyond the array by 5: %d",p);

   return 0;
}
```

For pointer p, what is the meaning of p = p+1, and p=p-1?



```

 Address of int
 6684160
 6684164
 6684168
 6684172

 p: 6684160
 p after add 1: 6684164
 p after add 1 and after add 2: 6684172
 p after add 1 and after add 2 and after -1: 6684168

 p: 6684172
 Address beyond the array by 1: 6684176

 p: 6684172
 Address beyond the array by 5: 6684192
--------------------------------
Process exited after 11.9 seconds with return value 0
Press any key to continue . . .
```

```
# include <stdio.h>

main()
{

  int   a[4], j;
  int *p;

  printf("\n\n Address of int");
  for (j=0;j<4;j++) printf("\n %d",&a[j]);

  p=a;
  printf("\n\n p: %d",p);

  p=p+1;
  printf("\n p after add 1: %d",p);

  p=p+2;
  printf("\n p after add 1 and after add 2: %d",p);

  p=p-1;
  printf("\n p after add 1 and after add 2 and after -1: %d",p);

  // ---- not to do these
   p=&a[3];
   printf("\n\n p: %d",p);
   p=p+1;
   printf("\n Address beyond the array by 1: %d",p);

   p=&a[3];
   printf("\n\n p: %d",p);
   p=p+5;
   printf("\n Address beyond the array by 5: %d",p);

  return 0;
}
```

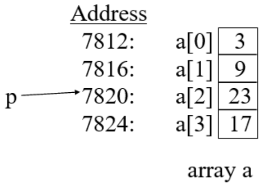
Source Code

By now you should know that all programming languages are artificial. They are all man-made.

Its meaning depends on the definition. Its actual meaning depends on the implementation by the compiler. All compilers are also man-made.

C takes into account the sizes of the array elements. Thus expressions such as *(p + 1) and *(p - 1) will work as expected regardless of how many bytes are occupied by each array element. E.g., each `char` requires 1 bytes, each `int` requires 4 bytes, each `float` requires 4 bytes.

- If `p` is pointing to integer `a[2],` what is the effect of `p = p + 1;` and `p=p-1;` ?



Address		
7812:	a[0]	3
7816:	a[1]	9
7820:	a[2]	23
7824:	a[3]	17

array a

5.2 The Qualifiers `const` and `volatile`

`const int b;`

means `b` is non-modifiable.

E.g.,

```
int add (const int a, const int b)
{
   return a+b;
}
```

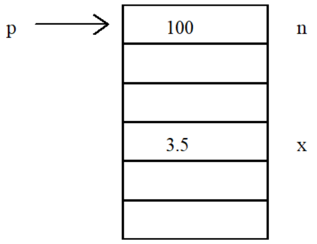
In the above function `a=2`; or `b=6`; will not be allowed as `a` and `b` are constants.

`volatile int n;`

The `volatile` qualifier states that the value of an object can be changed. All objects are volatile by default, unless they are specified as `const` or they are located in ROM.

5.3 Using the Address-Of Operator

```
int n = 100;
double x = 3.5;
int *p = &n;
```



`&n` has type `pointer-to-int (the address of n)`

`&x` has type `pointer-to-double (the address of x)`

`p` is initialized to point to `n`. Thus, `*p` and `n` name the same object.

```
printf("%d %d %lf",*p,n,x);
```

prints `100 100 3.5`

Immobilize a Pointer

```
int a=1, b=10;
int * const q=&a;
```

- `q=&b` is ***not*** allowed
- But the content where pointer `q` is pointing can be changed. E.g., `*q= 126;`

Disable the Change of Content

```
int a=1, b=10;
const int *p=&b;
```

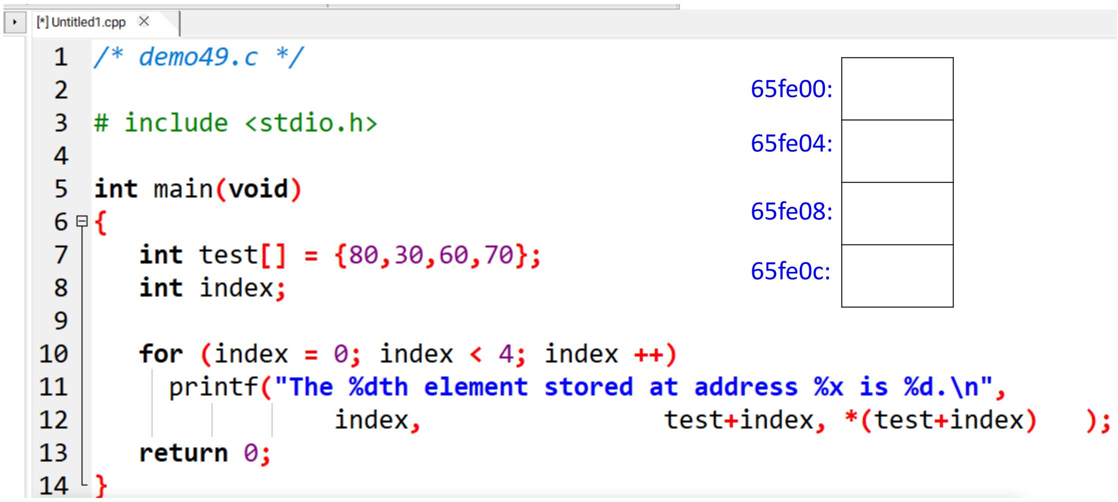
- `*p = 99` is ***not*** allowed, but `b=99;` is allowed
- `p=&a` is allowed

5.4 Pointers and Arrays

We can use the name of an array to initialize a pointer to the address of the first element of the array. For example, after

```
int list[100];
int *p=list;
int *q;
q = list;    // or q= &list[0];
```

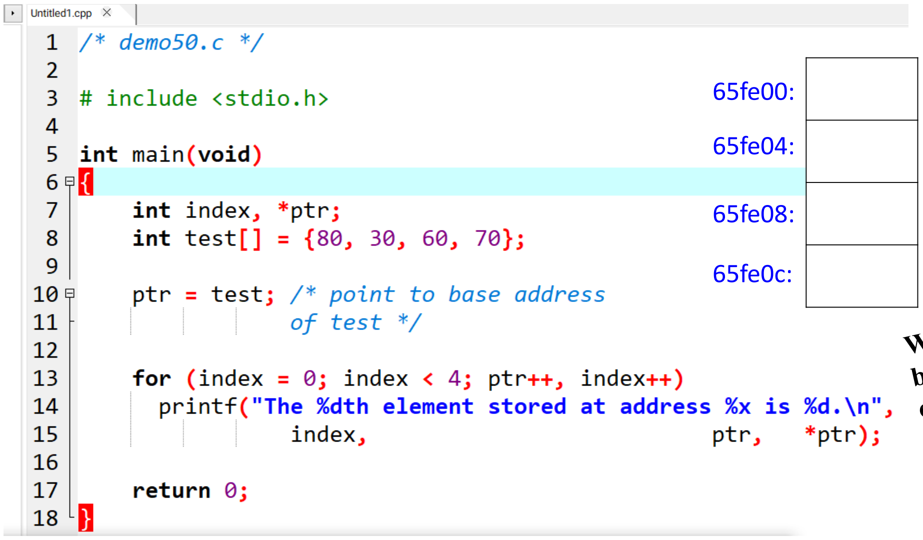
both p and q point to the first element of `list`.



```
/* demo49.c */

# include <stdio.h>

int main(void)
{
    int test[] = {80,30,60,70};
    int index;

    for (index = 0; index < 4; index ++)
      printf("The %dth element stored at address %x is %d.\n",
                 index,                  test+index, *(test+index)   );
    return 0;
}
```

```
/* demo49.c */

# include <stdio.h>

int main(void)
{
   int test[] = {80,30,60,70};
   int index;

   for (index = 0; index < 4; index ++)
     printf("The %dth element stored at address %x is %d.\n", index,
     test+index, *(test+index)   );
   return 0;
}
```

Source Code



Untitled1.cpp

```
/* demo50.c */

# include <stdio.h>

int main(void)
{
    int index, *ptr;
    int test[] = {80, 30, 60, 70};

    ptr = test; /* point to base address
                   of test */

    for (index = 0; index < 4; ptr++, index++)
      printf("The %dth element stored at address %x is %d.\n",
                   index,                          ptr,   *ptr);

    return 0;
}
```

What is the difference between two consecutive addresses ? Why ?

```
/* demo50.c */

# include <stdio.h>

int main(void)
{
    int index, *ptr;
    int test[] = {80, 30, 60, 70};

    ptr = test; /* point to base address
                         of test */

    for (index = 0; index < 4; ptr++, index++)
      printf("The %dth element stored at address %x is %d.\n",
                index,                  ptr,   *ptr);

    return 0;
}
```

Source Code

5.5 Pointers and Function Arguments

Pointer variables and the indirection and subscripting operators provide us with the means to utilize addresses passed to functions. For example,

```
int m = 10, n = 20;
swap(&m, &n);
printf ("\n %d %d", m, n);
```

the value of m will be 20 and that of n will be 10. Because swap() changes the values of m and n, their addresses (rather than their values) must be passed as arguments to the function swap.

```
void swap(int *p, int *q)
{
  int temp;
  temp = *p;
  *p = *q;
  *q = temp;
}
```

What if swap2 (m, n) is invoked ?

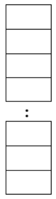
```
int m = 10, n = 20;
swap2(m, n);
printf ("\n %d %d", m, n);
```

```
void swap2 (int p, int q)
{
  int temp;
  temp = p;
  p = q;
  q = temp;
}
```

Example,

```
int list[50];
zero(list, 50);
```

will assign 0 to the first 50 elements of list. The first argument is passed as an address of type pointer-to-int:

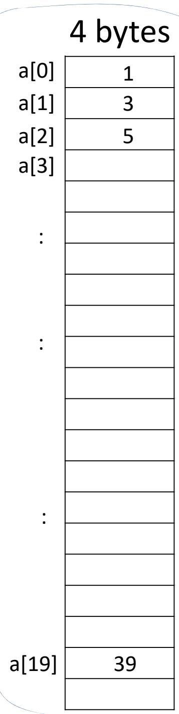


```
void zero(int *p, int count)
{
  int i;
  for (i = 0; i < count; i++)
    p[i] = 0;
}
```

We can also use array notation for formal arguments corresponding to arrays.

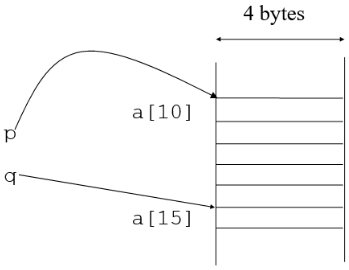
For example, the zero function could have been declared by void zero(int p[], int count);.

```
# include <stdio.h>

int sum(const int *p, int n)
{
  int i;
  int total = 0;

  for (i = 0; i < n; i++)
  {
    total += *p;
    p++;
  }

  return total;
}
main ()  /* for illustration */
{
   int a[20];
   int i, all;

   for (i=0;i <20; i++) a[i]=2*i+1;

   all = sum(a,20);
   printf ("\n Sum of 20 numbers is %d.",
                              all);
   return 0;
}
```



- We can subtract pointers that point to elements of the same array; the result is an integer equal to the difference of the corresponding array subscripts (not number of bytes).

Suppose a is an integer array. If p points to a[10] and q points to a[15], then q - p has the value of 5, which is the number of elements (not bytes) from a[10] to a[14].

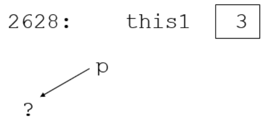


We have to add 1 if a[15] is to be included into the number of array elements.

E.g., How many numbers from 10 to 15 inclusive ?
Answer :

- The address of operator & may be used to direct a pointer to the location of non-array data. E.g.,

```
int this1=3, *p;
```



```
p = &this1; /* direct pointer p to point to
                   the location of this1 */
      /* read as : pointer p points
         to the address (location) of this1 */
What is the value for *p at this stage?
```

/* read as : What is the content where pointer p is pointing ? */

/* next instruction */

```
this1 = this1 +20;
What is the value for *p at this stage?
```

/* next instruction */

```
*p = 97;
What is the value for this1 at this stage?
```

5.6 Array Names and Pointers

- We can apply the sizeof operator to an array name to determine the number of bytes in the array.
- When an array name is used where a value is expected, the name is converted to the address of the first element of the array.

E.g., int a[5];

sizeof(a) returns 5*2=10 bytes.

&a[0] is the address of the first element of a, i.e., the address of a[0].

&a[0] is same as a.

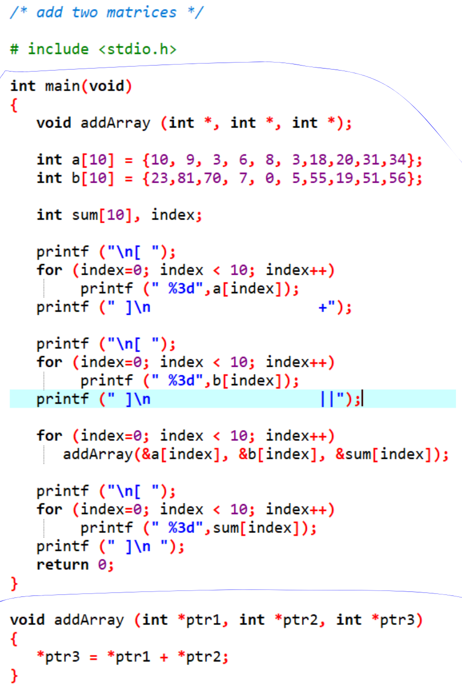
- The address of the array and the address of its first element are numerically equal.

- A pointer is the address of an object of a particular type. The type of a pointer specifies the type of the target object, the object designated by the address.

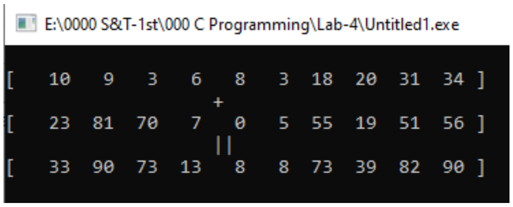
Typical pointer types are

pointer-to-int :	int *p;
pointer-to-long :	long *p;
pointer-to-double :	double *p;

and so on.

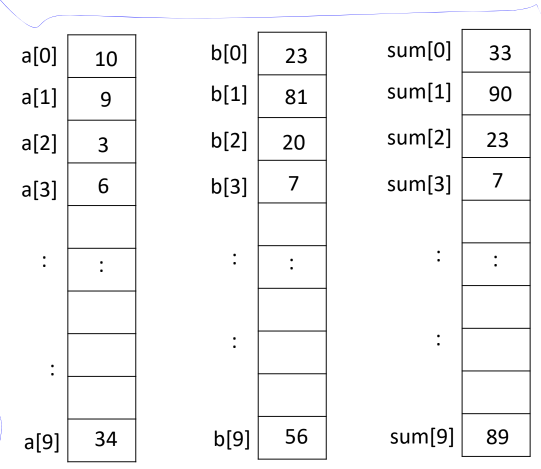


```c
/* add two matrices */

# include <stdio.h>

int main(void)
{
   void addArray (int *, int *, int *);

   int a[10] = {10, 9, 3, 6, 8, 3,18,20,31,34};
   int b[10] = {23,81,70, 7, 0, 5,55,19,51,56};

   int sum[10], index;

   printf ("\n[ ");
   for (index=0; index < 10; index++)
        printf (" %3d",a[index]);
   printf (" ]\n                    +");

   printf ("\n[ ");
   for (index=0; index < 10; index++)
        printf (" %3d",b[index]);
   printf (" ]\n                    ||");

   for (index=0; index < 10; index++)
     addArray(&a[index], &b[index], &sum[index]);

   printf ("\n[ ");
   for (index=0; index < 10; index++)
        printf (" %3d",sum[index]);
   printf (" ]\n ");
   return 0;
}

void addArray (int *ptr1, int *ptr2, int *ptr3)
{
   *ptr3 = *ptr1 + *ptr2;
}
```



E:\0000 S&T-1st\000 C Programming\Lab-4\Untitled1.exe

```
[   10   9   3   6   8   3  18  20  31  34 ]
                         +
[   23  81  70   7   0   5  55  19  51  56 ]
                         ||
[   33  90  73  13   8   8  73  39  82  90 ]
```



```c
# include <stdio.h>

int main(void)
{
  void addArray (int *, int *, int *);

  int a[10] = {10, 9, 3, 6, 8, 3,18,20,31,34};
  int b[10] = {23,81,70, 7, 0, 5,55,19,51,56};

  int sum[10], index;

  printf ("\n[ ");
  for (index=0; index < 10; index++)
      printf (" %3d",a[index]);
  printf (" ]\n                +");

  printf ("\n[ ");
  for (index=0; index < 10; index++)
      printf (" %3d",b[index]);
  printf (" ]\n                ||");

  for (index=0; index < 10; index++)
    addArray(&a[index], &b[index], &sum[index]);

  printf ("\n[ ");
  for (index=0; index < 10; index++)
      printf (" %3d",sum[index]);
  printf (" ]\n ");
  return 0;
}
```

Source Code

5.7 Multidimensional Arrays

A multidimensional array is one that requires more than one subscript to specify an element.

E.g.,

```
int table[3][4];
double book[9][6][4];
```

- The order in which the elements are stored in memory is determined by letting the right-most subscript vary most rapidly and the left-most subscript least rapidly.
- We initialize a multidimensional array by listing the elements in the order in which they are stored in memory.

```
int table[3][4] =   { {7, 9, 2, 5},
                      {8, 4, 6, 1},
                      {6, 5, 4, 2} };
```

or

```
int table[3][4] =    { 7, 9, 2, 5,
                       8, 4, 6, 1,
                       6, 5, 4, 2 };
```

After initialization, we have

```
table[0][0] = 7, table[0][1] = 9,
table[0][2] = 2, table[0][3] = 5,
table[1][0] = 8, . . , table[2][3] =2.
```

In general, a 2-D array with m rows and n columns can be visualized as an m*n 2-D matrix.

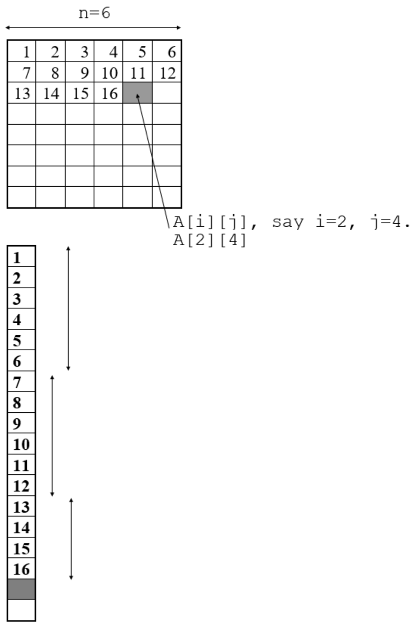
For example,

$$\text{table} = \begin{bmatrix} \text{table[0][0]} & \text{table[0][1]} & \ldots & \text{table[0][n-1]} \\ \text{table[1][0]} & \text{table[1][1]} & \ldots & \text{table[1][n-1]} \\ & & \vdots & \\ \text{table[m-1][0]} & \text{table[m-1][1]} & \ldots & \text{table[m-1][n-1]} \end{bmatrix}$$

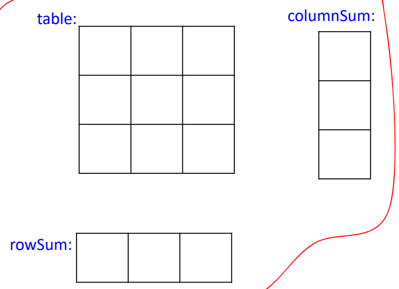
The individual element at row i column j is named by

`table[i][j]` or

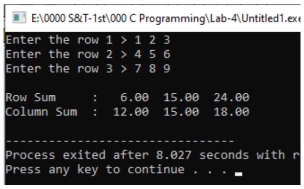
`*(&table[0][0] + i*n + j).`



```c
/* demo56a.c */
/* passing arrays to functions */

#include <stdio.h>

int nRows=3, nCols=3;

int main (void)
{
  void readInput (float [3][3]);
  void sumRowAndSumColumn (float [3][3],
                           float [3],float [3]);
  void writeOutput(float [3],
                           float [3]);
  float table[3][3], rowSum[3], columnSum[3];

  readInput (table);
  sumRowAndSumColumn (table, rowSum, columnSum);
  writeOutput(rowSum, columnSum);
  return 0;
}
```



```c
void readInput (float table[3][3])
{
  int i,j;

  for (i=0; i<nRows; i++)
  {
    printf ("Enter the row %d > ",i+1);
    for (j=0; j<nCols; j++)
       scanf("%f", &table[i][j]);
  }
}
```


E:\0000 S&T-1st\000 C Programming\Lab-4\Untitled1.exe

```
Enter the row 1 > 1 2 3
Enter the row 2 > 4 5 6
Enter the row 3 > 7 8 9

Row Sum     :   6.00  15.00  24.00
Column Sum  :  12.00  15.00  18.00

--------------------------------
Process exited after 8.027 seconds with r
Press any key to continue . . .
```

```c
void sumRowAndSumColumn (float
        table[3][3],           float rowSum[3],
        float columnSum[3])
{
  int i,j;
  for (i=0; i<nRows; i++)
  {
    rowSum[i] =0;
    for (j=0; j<nCols; j++)
         rowSum[i] += table[i][j];
  }
  for (j=0; j<nCols; j++)
  {
    columnSum[j] = 0;
    for (i=0; i<nRows; i++)
          columnSum[j] += table[i][j];
  }
}

void writeOutput(float    rowSum[3],   float columnSum[3])
{
  int i, j;
  printf ("\nRow Sum    :");
  for (i=0; i<nRows;i++)
    printf(" %6.2f",rowSum[i]);

  printf ("\nColumn Sum  :");
  for (j=0; j<nCols;j++)
    printf(" %6.2f",columnSum[j]);

  printf ("\n");
}
```

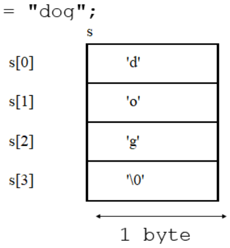
Source Code

5.8 Strings

- Strings are stored in memory as arrays of `char` values.
- A string terminator of escape sequence `\0` is appended to the end of each string.
- Therefore, the number of array elements must be one more than the number of characters, to provide room for the terminating null character.

E.g.,

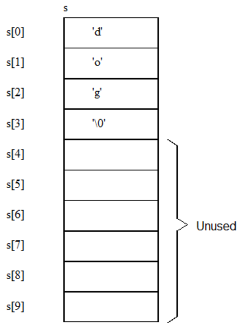
```c
char s[4];
char s[4] = "dog";
```



`char s[4] = "dog";`
is equivalent to

`char s[4] = {'d', 'o', 'g', '\0'};`
where the initialization is done one entry by one.

```
char s[10] = "dog";
```



`printf("%s",s);` and `printf("dog");`

literal

both cause the computer to print `dog` on the monitor screen. In each case, a pointer to the beginning of the string is passed to the `printf` function.

The pointer has type `char *` or `pointer-to-char`.

```
printf ( )
```

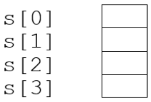
Array names and string literals can also be used to initialize and assign values to pointers. Thus

```
char s[4]= "dog";
char *p = s;       /* s is an array name */
```

initializes `p` to point to the first character of `s`, and

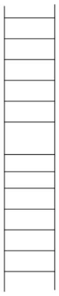
```
char *p = "dog";  /*  "dog" is a literal */
```

initializes `p` to point to the first character of `"dog"`.



The statement `printf("%s", p);` also causes the computer to print `dog` in the above two declarations and initialization.

5.9 Functions for String Processing



// strcopy.c

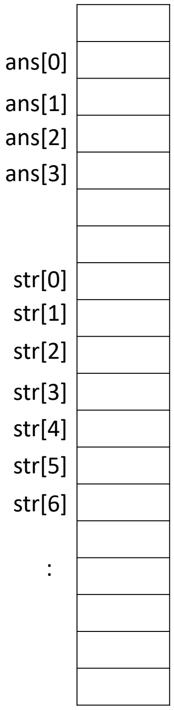
```
#include <string.h>
#include <stdio.h>

int main (void)
{
  char str1[] = "alpha";
  char str2[] = "beta";
  char *p;

  p = strcpy(str1, str2);

  printf("%s\n",p);
            /* beta will be printed */
  return 0;
}
```

How about

```
 printf("%s\n",str1);
 printf("%s\n",str2);
```

```
#include <string.h>
#include <stdio.h>

int main (void)
{
  char ans[] = "cat";
  char str[20];

  printf("      is afraid of dog ? ");

  printf("What is your answer?\n");
  scanf("%s", str);

  if(!strcmp(ans,str))
    printf("\nCorrect!");
  else
    printf("\nWrong!");

  return 0;
}
```

```
if( strcmp(ans,str)==0 )
    printf("\nCorrect!");
else
    printf("\nWrong!");
```



5.10 Input and Output of Characters and Strings

```
char this1 [50];
scanf ("%10c",  this1);
```

is to scan exactly 10 characters (white spaces are counted) without terminator. The program will wait until 10 characters have been entered.

```
char a[10], b[10], c[10], d[10];
scanf ("%s%s%s%s", a,b,c,d);
```

and the input line is `Now is the time.`

Results

```
  a : "Now\0"
  b : "is\0"
  c : "the\0"
  d : "time.\0"
```

```
char s[26];
scanf("%10s", s);
```

is to scan up to 10 characters or, white space is encountered at less than 10 characters.

e.g., If input is `2233445566778899`,

```
      s = "2233445566\0"

      s[0] ... s[9]    s[10]
       10 elements     11th element
```

e.g., If input is `223        3445566778899`,

```
      s = "223\0"
```

3+1=4 elements

```
char s[26];
scanf("%25s", s);
```

the array `s` cannot overflow because at most 25 characters can be read. Note that, because of the terminating null character, `s` must have 26 elements to accommodate a 25-character string.

How about

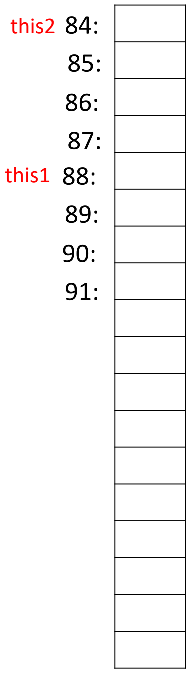
```
    char s[26];
    scanf("%26s", s);
```

And

```
    char s[26];
    scanf("%27s", s);
```

// string.c

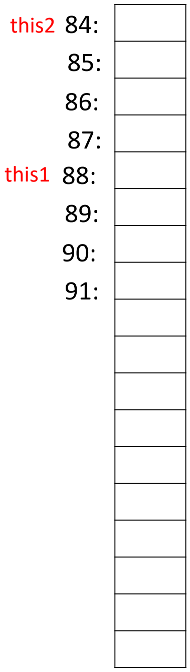
```
# include <stdio.h>
main()
{
  char this1 [4] = {'a','b','c','\0'};
  char this2 [4] = {'x','y','z','\0'};
  printf ("\n First address of this2:%d",&this2[0]);
  printf ("\n First address of this1:%d",&this1[0]);

  printf ("\n this2: %s",this2);
  printf ("\n this1: %s",this1);

  printf ("\n\n Enter this2:");
  scanf ("%s", this2);

  printf ("\n this2: %s",this2);
  printf ("\n this1: %s",this1);

  printf ("\n this1[0]: %c",this1[0]);
  printf ("\n this1[1]: %c",this1[1]);
  printf ("\n this1[2]: %c",this1[2]);
  printf ("\n this1[3]: %c",this1[3]);

  return 0;
}
```



```
First address of this2:6684184
First address of this1:6684188
this2: xyz
this1: abc

Enter this2:1234567890123

this2: 1234567890123
this1: 567890123
this1[0]: 5
this1[1]: 6
this1[2]: 7
this1[3]: 8
--------------------------------
```

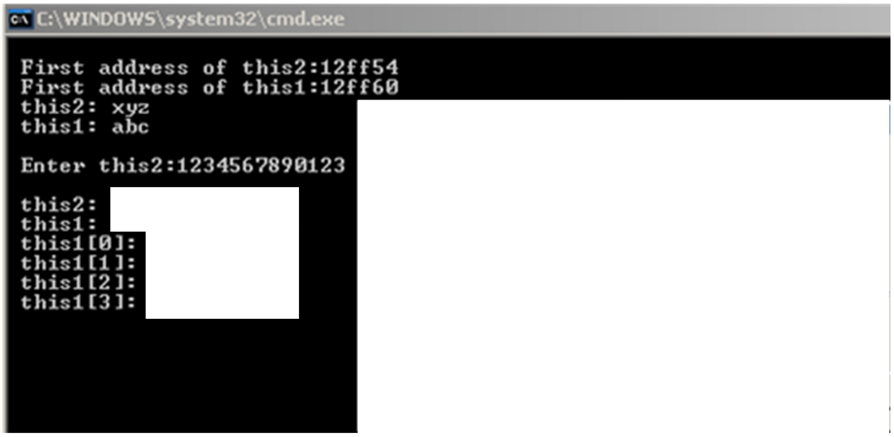
// string.c

```
# include <stdio.h>
main()
{
  char this1 [4] = {'a','b','c','\0'};
  char this2 [4] = {'x','y','z','\0'};
  printf ("\n First address of this2:%d",&this2[0]);
  printf ("\n First address of this1:%d",&this1[0]);

  printf ("\n this2: %s",this2);
  printf ("\n this1: %s",this1);

  printf ("\n\n Enter this2:");
  scanf ("%s", this2);

  printf ("\n this2: %s",this2);
  printf ("\n this1: %s",this1);

  printf ("\n this1[0]: %c",this1[0]);
  printf ("\n this1[1]: %c",this1[1]);
  printf ("\n this1[2]: %c",this1[2]);
  printf ("\n this1[3]: %c",this1[3]);

  return 0;
}
```



this2	ff54	x
	55	y
	56	z
	57	‘\0’
	58	
	59	
	5a	
	5b	
	5c	
	5d	
	5e	
	5f	
this1	60	a
	61	b
	62	c
	63	‘\0’
	64	
	65	

What if??



```
First address of this2:12ff54
First address of this1:12ff60
this2: xyz
this1: abc

Enter this2:1234567890123

this2:
this1:
this1[0]:
this1[1]:
this1[2]:
this1[3]:
```