Chapter 5 Pointer and Array

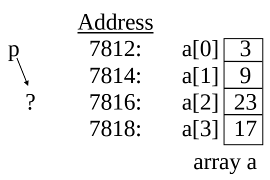
Ground Rules

- 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;`



`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`

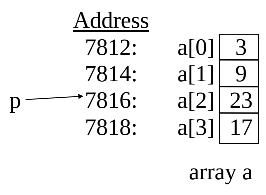
What is `p+4` ?

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 2 bytes, each `float` requires 4 bytes.

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



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;
    }
```

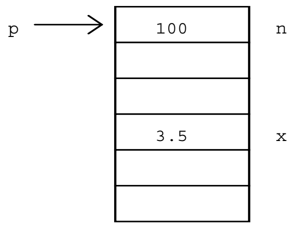
In the above function `a=2`; or `b=6`; will not be allowed as `a` and `b` are constants.

`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

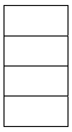
5.4 Pointers and Arrays

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

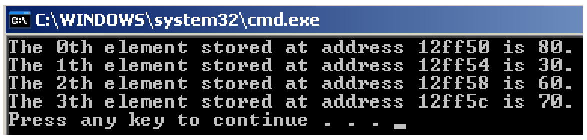
```
int list[100];
int *p=list;
int *q;
q = list;
```

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;
}
```

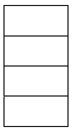


Screen Output:

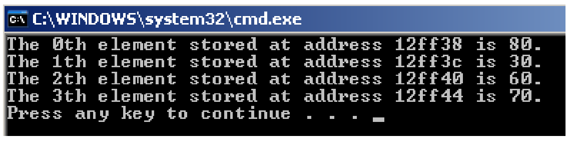


```
The 0th element stored at address 12ff50 is 80.
The 1th element stored at address 12ff54 is 30.
The 2th element stored at address 12ff58 is 60.
The 3th element stored at address 12ff5c is 70.
Press any key to continue . . . _
```

```
/* 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;
}
```



Screen Output:



```
The 0th element stored at address 12ff38 is 80.
The 1th element stored at address 12ff3c is 30.
The 2th element stored at address 12ff40 is 60.
The 3th element stored at address 12ff44 is 70.
Press any key to continue . . . _
```

What is the difference between two consecutive addresses ? Why ?

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 ?

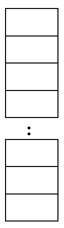
```
int m = 10, n = 20;
swap2(m, n);
printf ("\n %d %d", m, n);
```

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

Example,

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

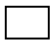
will assign 0 to the first 50 elements of `list`. The first argument is passed as an address of type `pointer-to-int`:



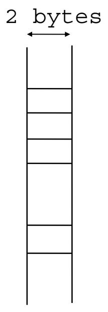
```
void zero(int *p, int count)
{
  int i;
  for (i = 0; i < count; i++)
    p[i] = 0;
}
```

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);`.

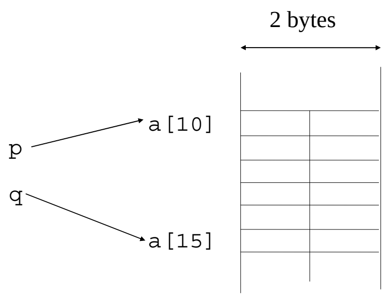


```
# include <stdio.h>

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]` .

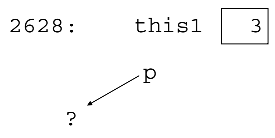


We have to add 1 if `a[15]` is to be included.

E.g., How many numbers from 10 to 15 inclusive ?
Answer : 15-10 + 1 = 6 numbers

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

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



```
p = &this1; /* direct p to point to
                the location of this1 */
       /* read as : pointer p points
          to the address (location)
          of this1 */
```

```
*p = what ?
```

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

/* next instruction */

```
this1 = this1 +20;
 *p = what again ?
```

/* next instruction */

```
*p = 97;
this1 = what ?
```

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.

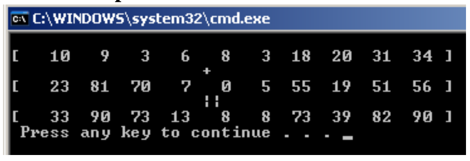
  Typical pointer types are

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

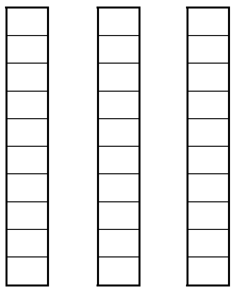
  and so on.

```
/* demo53.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;
}
```

Screen Output:



```
[   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 ]
 Press any key to continue . . .
```



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.

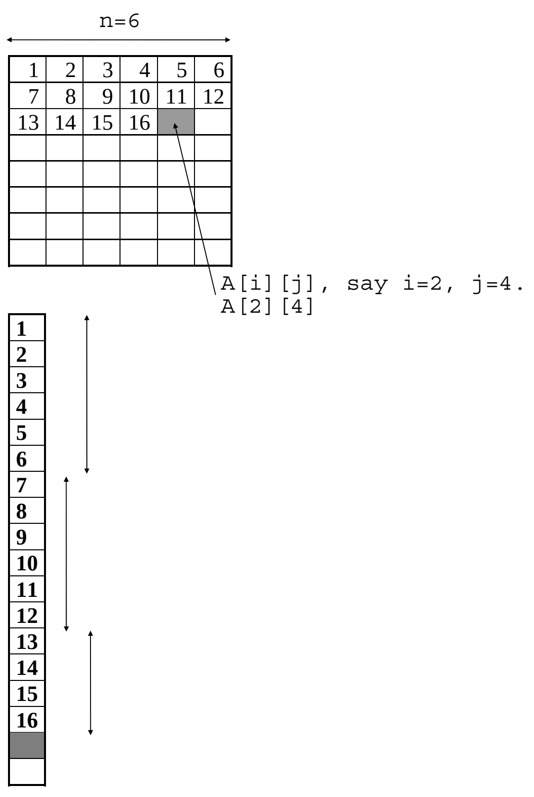
For example,

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

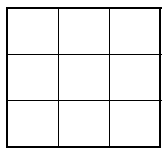
The individual element at row i column j is named by

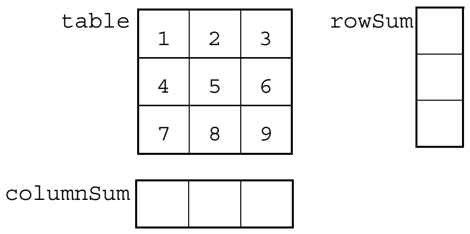
`table[i][j]` or

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

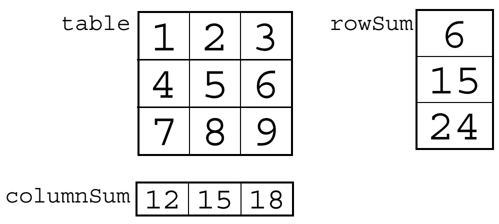


```
/* demo56a.c */
/* passing arrays to functions */

#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;
}


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]);
   }
}
```



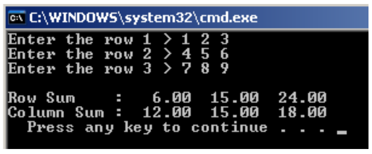


```
void sumRowAndSumColumn (float
table[3][3],                float rowSum[3],
        float columnSum[3])
{
   int i,j;

   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");

}
```

Screen Ouput:



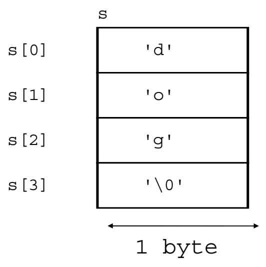
```
Enter the row 1 > 1 2 3
Enter the row 2 > 4 5 6
Enter the row 3 > 7 8 9

Row Sum     :   6.00  15.00  24.00
Column Sum :  12.00  15.00  18.00
  Press any key to continue . . .
```

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.,

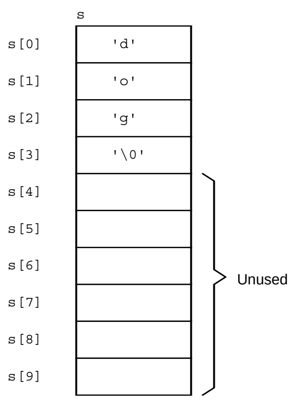
```
char s[4];
char s[4] = "dog";
```



`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 ( )`

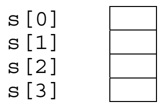
Array names and string literals can also be used to initialize and assign values to pointers. Thus

```
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 */
```

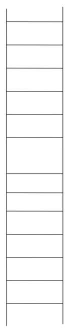
initializes `p` to point to the first character of `"dog"`.



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

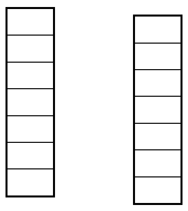


```
#include <string.h>
#include <stdio.h>

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);
```

// strcmp.c



```
#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

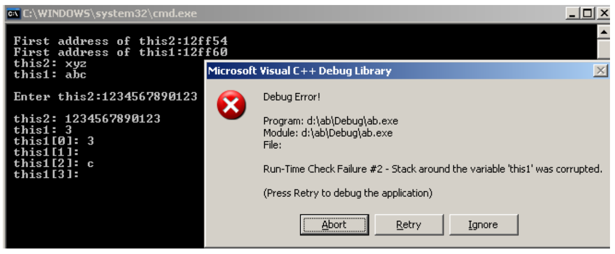
```
char s[26];
scanf("%26s", s);
```

And

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

This program is interesting:

```
// 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:%x",&this2[0]);
  printf ("\n First address of this1:%x",&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;
}
```


C:\WINDOWS\system32\cmd.exe
Microsoft Visual C++ Debug Library
Abort
Retry
Ignore

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

Enter this2:1234567890123

this2: 1234567890123
this1: 3
this1[0]: 3
this1[1]:
this1[2]: c
this1[3]:
```

Debug Error!

Program: d:\ab\Debug\ab.exe
Module: d:\ab\Debug\ab.exe
File:

Run-Time Check Failure #2 - Stack around the variable 'this1' was corrupted.

(Press Retry to debug the application)

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	

Chapter 6 Types and Conversions

6.1 Type Definitions

```
typedef int whole_num;
whole_num count;    /* int count; */
```

```
typedef float real;
real inches;
```

Usually `typedef` is used in more complicated data type, such as struct.

6.2 Type `size_t` and the `sizeof` Operator

- A C implementation defines a number of `typedef` names in header files.

- Each `typedef` name corresponds to a particular purpose that a type can serve. For example, `size_t` is an unsigned integer type used to represent the number of bytes in a region of memory. Depending on the computer hardware, type `unsigned` or `unsigned long` might reasonably be used for this job.

- The `sizeof` operator can be applied directly to a type rather than an expression, but the type must be enclosed in parentheses. `sizeof` returns the number of bytes used by the type.

- For example, `sizeof(unsigned long)` yields the size of an `unsigned long` value.

6.3 Enumerated Types

- An enumeration specifies an integer type and defines constants to represent values of the type.
- The constants are called enumeration constants and (like character constants) have type `int`. We can use an enumeration to declare variables with the enumerated type. We can also test the values of the variables with expressions.

For example, the following statements

```
enum workday {MON=1, TUE, WED, THUR, FRI};
int day;

for (day=MON; day <= FRI; day++)
  printf("%d ",day);
```

produce the printout `1   2   3   4   5`

6.4 Type Casts

- Type casts allow us to request conversions explicitly wherever they may be needed.

- A type cast is an operator formed by enclosing a type designation in parentheses, as `(int)`, `(float)`, and `(unsigned long)`.

For example,

```
float x;
int n;

n = (int)(x + 0.5);
```

assign `n` the value of `x` rounded to the nearest integer.

```
long n;

n = (long)(32760 + 8);
```

6.5 Format Strings for `scanf()`

- For `scanf()`, a conversion specification gives the format of a value to be read; the corresponding argument gives the address of the variable in which the value is to be stored.
  Example: `scanf("%d",&a);`

- Literal characters in a `scanf()` format string are matched with corresponding characters in the input. Input characters that match literal characters are read and discarded.
  Example: `scanf("%f",&a);`
  Input: ~~`12.34`~~ `93`

- If a literal character (other than a space) fails to match the corresponding input character, `scanf()` returns immediately without reading any more input or storing any more values.
  Example, `scanf("***$%lf",&x);`
  requires the input to begin with the characters ***$. Thus ***$25.99 is acceptable but **$25.99 will cause `scanf()` to return before the number is read and stored.

  Example, `scanf("%*d%lf",&x);`
  reads and discards an `int` value, then reads a double value into x. So, if the input data is
  `50 2.99`
  x will have a value of `2.99`.

Chapter 7 Macros

- When a function is called, a certain amount of time is required to pass arguments to the function, transfer control to it, allocate memory for its variables, return its value, and transfer control back to the code following the function call.

- This overhead can be eliminated by replacing a function call by an inline code.

- Function macros provide a simple means of replacing certain function calls by inline code. Function macros are defined like other macros except that one or more formal arguments are listed in parentheses after the macro name.

- Formal arguments are used in the expression that defines the macro.

Example 1

Consider the following function:

```
float sumsqr (float x, float y)
{
  float w;
  w= x*x + y*y;
  return w;
}
```

The function `sumsqr` can be replaced by a macro as follows:

```
#define sumsqr(x, y)  x*x + y*y
```

The macro has two formal arguments, `x` and `y`. A macro call specifies which text (expression) will **replace** which formal argument when the call is expanded.

If we write

```
u = sumsqr(a, 2.75);
```

then `a` will replace `x`, and `2.75` will replace `y`, and the statement will expand to

```
u = a*a + 2.75*2.75;
```

i.e., x is replaced by a, and
    y is replaced by 2.75

Example 2

```
#define sumsqr(x, y) ((x)*(x) + (y)*(y))
```

Now , `sumsqr(a + b, c)` expands to

```
((a + b)*(a + b) + (c)*(c))
```

and `2.0*sumsqr(a, b)` expands to

```
2.0*((a)*(a) + (b)*(b))
```

The expanded expressions yield the expected results even though they contain a few more parentheses than we would normally write.

The preprocessor recognizes a call to a function macro only if the name of the macro is followed by a left parenthesis.

```
/*  demo67.c */

#include <stdio.h>
#define cube(x) ((x) * (x) * (x))

int main (void)
{
   printf( "The cube of 3.2 is %.3f \n",
                cube(3.2));
   return 0;
}
```

Screen Output:

```
The cube of 3.2 is 32.768
```