Notice that the string **"M=%d\n"** is left unchanged.

A macro definition can include more than a simple constant value. It can include expressions as well. Following are valid definitions:

```
#define     AREA      5 * 12.46
#define     SIZE      sizeof(int) * 4
#define     TWO-PI    2.0 * 3.1415926
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. Consider the evaluation of the equation

```
ratio = D/A;
```

where D and A are macros defined as follows:

```
#define  D  45 - 22
#define  A  78 + 32
```

The result of the preprocessor's substitution for D and A is:

```
ratio = 45-22/78+32;
```

This is certainly different from the expected expression

```
(45 - 22)/(78+32)
```

Correct results can be obtained by using parentheses around the strings as:

```
#define  D  (45 - 22)
#define  A  (78 + 32)
```

It is a wise practice to use parentheses for expressions used in macro definitions.

As mentioned earlier, the preprocessor performs a literal text substitution whenever the defined name occurs. This explains why we cannot use a semicolon to terminate the #define statement. This also suggests that we can use a macro to define almost anything. For example, we can use the definitions

```
#define     TEST      (45 - 22)
#define     AND
#define     PRINT     printf("Very Good. \n");
```

to build a statement as follows:

```
TEST AND PRINT
```

The preprocessor would translate this line to

```
if(x>y) printf("Very Good.\n");
```

Some tokens of C syntax are confusing or are error-prone. For example, a common programming mistake is to use the token = in place of the token == in logical expressions. Similar is the case with the token &&.

Following are a few definitions that might be useful in building error free and more readable programs:

```
#define     EQUALS    ==
#define     AND       &&
```

```
#define     OR          ||
#define     NOT_EQUAL   !=
#define     START       main() {
#define     END         }
#define     MOD         %
#define     BLANK_LINE  printf("\n");
#define     INCREMENT   ++
```

An example of the use of syntactic replacement is:

```
START
... ...
... ...
if(total EQUALS 240 AND average EQUALS 60)
INCREMENT count;
... ...
... ...
END
```

Macros with Arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

#define identifier(f1, f2, ... fn) string

Notice that there is no space between the macro *identifier* and the left parentheses. The identifiers f1, f2, ...,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

```
#define     CUBE(x)     (x*x*x)
```

If the following statement appears later in the program

```
volume = CUBE(side);
```

Then the preprocessor would expand this statement to:

```
volume = (side * side * side );
```

Consider the following statement:

```
volume = CUBE(a+b);
```

This would expand to:

```
volume = (a+b * a+b * a+b);
```

which would obviously not produce the correct results. This is because the preprocessor performs a blind test substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string*.
Example:

```
#define        CUBE(x)        ( (x) * (x) *(x) )
```

This would result in correct expansion of **CUBE(a+b)** as:

```
volume = ( (a+b) * (a+b) * (a+b));
```

Remember to use parentheses for each occurrence of a formal argument, as well as the whole *string*.

Some commonly used definitions are:

```
#define        MAX(a,b)        (((a) > (b)) ? (a) : (b))
#define        MIN(a,b)        (((a) < (b)) ? (a) : (b))
#define        ABS(x)          (((x) > 0) ? (x) : (-(x)))
#define        STREQ(s1,s2)    (strcmp((s1,) (s2)) == 0)
#define        STRGT(s1,s2)    (strcmp((s1,) (s2)) > 0)
```

The argument supplied to a macro can be any series of characters. For example, the definition

```
#define PRINT(variable, format)   printf("variable = %format \n", variable)
```

can be called-in by

```
PRINT(price x quantity, f);
```

The preprocessor will expand this as

```
printf( "price x quantity = %f\n", price x quantity);
```

Note that the actual parameters are substituted for formal parameters in a macro call, although they are within a string. This definition can be used for printing integers and character strings as well.

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions.

```
#define        M               5
#define        N               M+1
#define        SQUARE(x)       ( (x) * (x) )
#define        CUBE(x)         (SQUARE (x) * (x))
#define        SIXTH(x)        (CUBE(x) * CUBE(x))
```

The preprocessor expands each **#define** macro, until no more macros appear in the text. For example, the last definition is first expanded into

```
((SQUARE(x) * (x)) * (SQUARE(x) * (x)))
```

Since **SQUARE (x)** is still a macro, it is further expanded into

```
( ( ((x)*(x)) * (x) ) * ( ((x) * (x)) * (x)) )
```

which is finally evaluated as x^6.

Macros can also be used as parameters of other macros. For example, given the definitions of M and N, we can define the following macro to give the maximum of these two:

```
#define MAX(M,N)  (( (M) > (N) ) ? (M) : (N))
```

Macro calls can be nested in much the same fashion as function calls. Example:

```
#define     HALF(x)     ( (x)/2.0)
#define     Y           HALF(HALF(x))
```

Similarly, given the definition of MAX(a,b) we can use the following nested call to give the maximum of the three values x,y, and z:

```
MAX (x, MAX(y,z))
```

Undefining a Macro

A defined macro can be undefined, using the statement

#undef *identifier*

This is useful when we want to restrict the definition only to a particular part of the program.

14.3 FILE INCLUSION

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive.

#include "filename"

where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <filename>

without double quotation marks. In this case, the file is searched only in the standard directories.

Nesting of included files is allowed. That is, an included file can include other files. However, a file cannot include itself.

If an included file is not found, an error is reported and compilation is terminated.

Let use assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the program as:

```
#include   <stdio.h>
#include   "SYNTAX.C"
#include   "STAT.C"
#include   "TEST.C"
#define    M       100
main ()
{
      ------------
      ------------
      ------------
}
```

14.4 COMPILER CONTROL DIRECTIVES

While developing large programs, you may face one or more of the following situations:

1. You have included a file containing some macro definitions. It is not known whether a particular macro (say, TEST) has been defined in that header file. However, you want to be certain that Test is defined (or not defined).
2. Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although certain lines of code must be different for each system.
3. You are developing a program (say, for sales analysis) for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.
4. Suppose you are in the process of testing your program, which is rather a large one. You would like to have print calls inserted in certain places to display intermediate results and messages in order to trace the flow of execution and errors, if any. Such statements are called 'debugging' statements. You want these statements to be a part of the program and to become 'active' only when you decide so.

One solution to these problems is to develop different programs to suit the needs of different situations. Another method is to develop a single, comprehensive program that includes all optional codes and then directs the compiler to skip over certain parts of source code when they are not required. Fortunately, the C preprocessor offers a feature known as *conditional compilation*, which can be used to 'switch' on or off a particular line or group of lines in a program.

Situation 1

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include   "DEFINE.H"
#ifndef    TEST
#define    TEST 1
#endif
... ...
```

DEFINE.H is the header file that is supposed to contain the definition of **TEST** macro. The directive.

#ifndef TEST

searches for the definition of **TEST** in the header file and *if not defined*, then all the lines between the **#ifndef** and the corresponding **#endif** directive are left 'active' in the program. That is, the preprocessor directive

 # define TEST is processed.

In case, the TEST has been defined in the header file, the **#ifndef** condition becomes false, therefore the directive **#define TEST** is ignored. Remember, you cannot simply write

 # define TEST 1

because if **TEST** is already defined, an error will occur.

Similar is the case when we want the macro **TEST** never to be defined. Looking at the following code:

```
... ... ...
#ifdef    TEST
#undef    TEST
#endif
... ...
... ...
```

This ensures that even if **TEST** is defined in the header file, its definition is removed. Here again we cannot simply say

#undef TEST

because, if TEST is not defined, the directive is erroneous.

Situation 2

The main concern here is to make the program portable. This can be achieved as follows:

```
... ...
... ...
main()
{
   ... ...
   ... ...
#ifdef IBM_PC
{
   ... ...
   ... ...                    code for IBM_PC
   ... ...
}
#else
{
```

```
    ... ...
    ... ...          code for HP machine
    ... ...
}
#endif
    ... ..
    ... ..
}
```

If we want the program to run on IBM PC, we include the directive

#define IBM_PC

in the program; otherwise we don't. Note that the compiler control directives are inside the function. Care must be taken to put the # character at column one.

The compiler complies the code for IBM PC if **IBM-PC** is defined, or the code for the HP machine if it is not.

Situation 3

This is similar to the above situation and therefore the control directives take the following form:

```
#ifdef    ABC
      group-A lines
#else
      group-B lines
#endif
```

Group-A lines are included if the customer **ABC** is defined. Otherwise, group-B lines are included.

Situation 4

Debugging and testing are done to detect errors in the program. While the Compiler can detect syntactic and semantic errors, it cannot detect a faulty algorithm where the program executes, but produces wrong results.

The process of error detection and isolation begins with the testing of the program with a known set of test data. The program is divided down and printf statements are placed in different parts to see intermediate results. Such statements are called debugging statements and are not required once the errors are isolated and corrected. We can either delete all of them or, alternately, make them inactive using control directives as:

```
... ...
... ...
#ifdef TEST
{
        printf("Array elements\n");
        for (i = 0; i< m; i++)
            printf("x[%d] = %d\n", i, x[i]);
```

Note that we have used a special processor operator **defined** along with **#if**. **defined** is a new addition and takes a *name* surrounded by parentheses. If a compiler does not support this, we can replace it as follows:

#if !defined	by	#ifndef
#if defined	by	#ifdef

Stringizing Operator

ANSI C provides an operator # called *stringizing operator* to be used in the definition of macro functions. This operator allows a formal argument within a macro definition to be converted to a string. Consider the example below:

```
#define sum(xy) printf(#xy " = %f\n", xy)
main()
{
   ... ...
   ... ...
   sum(a+b);
   ... ...
}
```

The preprocessor will convert the line

```
sum(a+b);
```

into

```
printf("a+b" "=%f\n", a+b);
```

which is equivalent to

```
printf("a+b =%f\n", a+b);
```

Note that the ANSI standard also stipulates that adjacent strings will be concatenated.

Token Pasting Operator

The token pasting operator ## defined by ANSI standard enables us to combine two tokens within a macro definition to form a single token. For example:

```
#define combine(s1,s2) s1 ## s2
main()
{
        ... ..
        ... ..
        printf("%f", combine(total, sales));
        ... ..
        ... ..
}
```

The preprocessor transforms the statement

```
printf("%f", combine(total, sales));
```

into the statement

```
printf("%f", totalsales);
```

Consider another macro definition:

```
#define print(i) printf("a" #i "=%f", a##i)
```

This macro will convert the statement

```
print(5);
```

into the statement

```
printf("a5 = %f", a5)
```

REVIEW QUESTIONS

14.1 Explain the role of the C preprocessor.
14.2 What is a macro and how is it different from a C variable name?
14.3 What precautions one should take when using macros with argument?
14.4 What are the advantages of using macro definitions in a program?
14.5 When does a programmer use **#include** directive?
14.6 The value of a macro name cannot be changed during the running of a program. Comment?
14.7 What is conditional compilation? How does it help a programmer?
14.8 Distinguish between **#ifdef** and **#if** directives.
14.9 Comment on the following code fragment:

```
#if 0
{
        line-1;
        line-2;
        ... ...
        ... ...
        line-n;
}
#endif
```

14.10 Identify errors, if any, in the following macro definitions:
(a) #define until(x) while(!x)
(b) #define ABS(x) (x > 0) ? (x) : (–x)
(c) #ifdef(FLAG)
#undef FLAG
#endif

Problem Analysis

Before we think of a solution procedure to the problem, we must fully understand the nature of the problem and what we want the program to do. Without the comprehension and definition of the problem at hand, program design might turn into a hit-or-miss approach. We must carefully decide the following at this stage;

What kind of data will go in,
What kind of outputs are needed, and
What are the constraints and conditions under which the program has to operate?

Outlining the Program Structure

Once we have decided what we want and what we have, then the next step is to decide how to do it. C as a structured language lends itself to a *top-down* approach. Top-down means decomposing of the solution procedure into tasks that form a hierarchical structure as shown in Fig. 15.1. The essence of the top-down design is to cut the whole problem into a number of independent constituent tasks, and then to cut the tasks into smaller subtasks, and so on, until they are small enough to be grasped mentally and to code easily. These tasks and subtasks can form the basis of functions in the program.

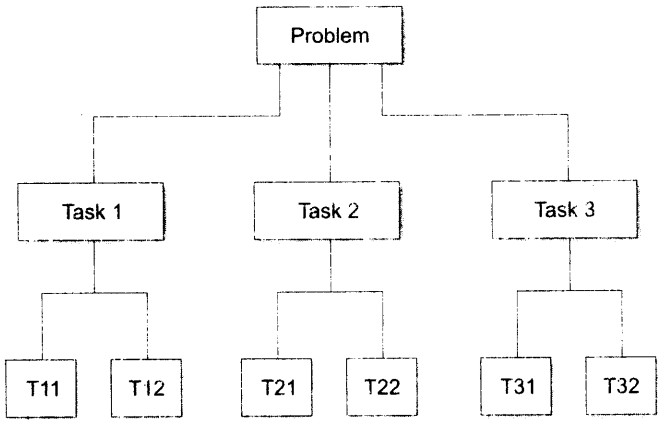

Fig. 15.1 *Hierarchical structure*

An important feature of this approach is that at each level, the details of the design of lower levels are hidden. The higher-level functions are designed first, assuming certain broad tasks of the immediately lower level functions. The actual details of the lower level functions are not considered until that level is reached. Thus the design of functions proceeds from top to bottom, introducing progressively more and more refinements.

This approach will produce a readable and modular code that can be easily understood and maintained. It also helps us classify the overall functioning of the program in terms of lower-level functions.

Algorithm Development

After we have decided a solution procedure and an overall outline of the program, the next step is to work out a detailed definite, step-by-step procedure, known as *algorithm* for each function. The most common method of describing an algorithm is through the use of *flowcharts*. The other method is to write what is known as *pseudocode*. The flow chart presents the algorithm pictorially, while the pseudocode describe the solution steps in a logical order. Either method involves concepts of logic and creativity.

Since algorithm is the key factor for developing an efficient program, we should devote enough attention to this step. A problem might have many different approaches to its solution. For example, there are many sorting techniques available to sort a list. Similarly, there are many methods of finding the area under a curve. We must consider all possible approaches and select the one, which is simple to follow, takes less execution time, and produces results with the required accuracy.

Control Structures

A complex solution procedure may involve a large number of control statements to direct the flow of execution. In such situations, indiscriminate use of control statements such as **goto** may lead to unreadable and uncomprehensible programs. It has been demonstrated that any algorithm can be structured, using the three basic control structure, namely, sequence structure, selection structure, and looping structure.

Sequence structure denotes the execution of statements sequentially one after another. Selection structure involves a decision, based on a condition and may have two or more branches, which usually join again at a later point. **if . . . .else** and **switch** statements in C can be used to implement a selection structure. Looping structure is used when a set of instructions is evaluated repeatedly. This structure can be implemented using **do**, **while**, or **for** statements.

A well-designed program would provide the following benefits:

1. Coding is easy and error-free.
2. Testing is simple.
3. Maintenance is easy.
4. Good documentation is possible.
5. Cost estimates can be made more accurately.
6. Progress of coding may be controlled more precisely.

15.3 PROGRAM CODING

The algorithm developed in the previous section must be translated into a set of instructions that a computer can understand. The major emphasis in coding should be simplicity and clarity. A program written by one may have to be ready by others later. Therefore, it should be readable and simple to understand. Complex logic and tricky coding should be avoided. The elements of coding style include:

- internal documentation.
- construction of statements.
- generality of the program.
- input/output formats.

Internal Documentation

Documentation refers to the details that describe a program. Some details may be built-in as an integral part of the program. These are known as *internal documentation*.

Two important aspects of internal documentation are, selection of meaningful variable names and the use of comments. Selection of meaningful names is crucial for understanding the program. For example,

area = breadth * length

is more meaningful than

```
a = b * l;
```

Names that are likely to be confused must be avoided. The use of meaningful function names also aids in understanding and maintenance of programs.

Descriptive comments should be embedded within the body of source code to describe processing steps.

The following guidelines might help the use of comments judiciously:

1. Describe blocks of statements, rather than commenting on every line.
2. Use blank lines or indentation, so that comments are easily readable.
3. Use appropriate comments; an incorrect comment is worse than no comment at all.

Statement Construction

Although the flow of logic is decided during design, the construction of individual statements is done at the coding stage. Each statement should be simple and direct. While multiple statements per line are allowed, try to use only one statement per line with necessary indentation. Consider the following code:

```
if(quantify>0){code = 0; quantity = rate;}
else { code = 1; sales = 0:)
```

Although it is perfectly valid, it could be reorganized as follows:

```
if(quantify>0)
{
    code = 0;
    quantify = rate;
}
else
{
    code = 1;
    sales = 0:
}
```

The general guidelines for construction of statements are:

1. Use one statement per line.
2. Use proper indentation when selection and looping structures are implemented.

3. Avoid heavy nesting of loops, preferably not more than three levels.
4. Use simple conditional tests; if necessary break complicated conditions into simple conditions.
5. Use parentheses to clarify logical and arithmetic expressions.
6. Use spaces, wherever possible, to improve readability.

Input/Output Formats

Input/output formats should be simple and acceptable to users. A number of guidelines should be considered during coding.

1. Keep formats simple.
2. Use end-of-file indicators, rather than the user requiring to specify the number of items.
3. Label all interactive input requests.
4. Label all output reports.
5. Use output messages when the output contains some peculiar results.

Generality of Programs

Care should be taken to minimize the dependence of a program on a particular set of data, or on a particular value of a parameter. Example:

```
for(sum = 0, i=1; i <= 10; i++)
    sum = sum + i;
```

This loop adds numbers 1,2, ...10. This can be made more general as follows:

```
sum =0;
for(i =m; i <=n; i = i+ step);
    sum = sum + i;
```

The initial value **m**, the final value **n**, and the increment size **step** can be specified interactively during program execution. When **m**=2, **n**=100, and **step** =2, the loop adds all even numbers up to, and including 100.

15.4 COMMON PROGRAMMING ERRORS

By now you must be aware that C has certain features that are easily amenable to bugs. Added to this, it does not check and report all kinds of run-time errors. It is therefore advisable to keep track of such errors and to see that these known errors are not present in the program. This section examines some of the more common mistakes that a less experienced C programmer could make.

Missing Semicolons

Every C statement must end with a semicolon. A missing semicolon may cause considerable confusion to the compiler and result in 'misleading' error messages. Consider the following statements:

```
a = x+y
b = m/n;
```

The compiler will treat the second line as a part of the first one and treat b as a variable name. You may therefore get an "undefined name" error message in the second line. Note that both the message and location are incorrect. In such situations where there are no errors in a reported line, we should check the preceding line for a missing semicolon.

There may be an instance when a missing semicolon might cause the compiler to go 'crazy' and to produce a series of error messages. If they are found to be dubious errors, check for a missing semicolon in the beginning of the error list.

Misuse of Semicolon

Another common mistake is to put a semicolon in a wrong place. Consider the following code:

```
for(i = 1; i<=10; i++);
        sum = sum + i;
```

This code is supposed to sum all the integers from 1 to 10. But what actually happens is that only the 'exit' value of i is added to the sum. Other examples of such mistake are:

```
1. while (x < Max);
   {

   }
2. if(T>= 200);
   grade = 'A';
```

A simple semicolon represents a null statement and therefore it is syntactically valid. The compiler does not produce any error message. Remember, these kinds of errors are worse than syntax errors.

Use of = Instead of = =

It is quite possible to forget the use of double equal sings when we perform a relational test. Example:

```
if(code = 1)
   count ++;
```

It is a syntactically valid statement. The variable code is assigned 1 and then, because code = 1 is true, the count is incremented. In fact, the above statement does not perform any relational test on code. Irrespective of the previous value of code, count ++: is always executed.

Similar mistakes can occur in other control statements, such as **for** and **while**. Such a mistake in the loop control statements might cause infinite loops.

Missing Braces

It is common to forget a closing brace when coding a deeply nested loop. It will be usually detected by the compiler because the number of opening braces should match with the closing ones. However, if we put a matching brace in a wrong place, the compiler won't notice the mistake and the program will produce unexpected results.

Another serious problem with the braces is, not using them when multiple statements are to be grouped together. For instance, consider the following statements: