```
if (value = product ( ) >= 100)
            tax = 0.05 * value;
```

The call **product ( )** returns the product of two numbers, which is compared to 100. If it is equal to or greater than 100, the relational test is true, and a 1 is assigned to **value**; otherwise a 0 is assigned. In either case, the only values **value** can take on are 1 or 0. This certainly is not what the programmer wanted.

The statement was actually expected to assign the value returned by **product( )** to **value** and then compare **value** with 100. If **value** was equal to or greater than 100, tax should have been computed, using the statement

```
tax = 0.05 * value;
```

The error is due to the higher precedence of the relational operator compared to the assignment operator. We can force the assignment to occur first by using parentheses as follows:

```
if(value = product()) >=100)
tax = 0.05 * value;
```

Similarly, the logical operators **&&** and || have lower precedence than arithmetic and relational operators and among these two, **&&** has higher precedence than ||. Try, if there is any difference between the following statements:

1. if(p > 50|| c > 50 && m > 60 && T > 180)
   x = 1;
2. if((p > 50|| c > 50) && m > 60 && T > 180)
   x = 1;
3. if((p > 50|| c > 50 && m > 60) && T > 180)
   x = 1;

Ignoring the Order of Evaluation of Increment/Decrement Operators

We often use increment or decrement operators in loops. Example

```
... ...
i = 0;
while ((c = getchar()) != '\n';
{
      string[i++] = c;
}
string[i-1] = '\n';
```

The statement **string[i++]** = c; is equivalent to :

```
string[i] = c;
i = i+1;
```

This is not the same as the statement **string[++i]** = c; which is equivalent to

```
i =i+1;
string[i] = c;
```

Forgetting to Declare Function Parameters

Remember to declare all function parameters in the function header.

Mismatching of Actual and Formal Parameter Types in Function Calls

When a function with parameters is called, we should ensure that the type of values passed, match with the type expected by the called function. Otherwise, erroneous results may occur. If necessary, we may use the *type* cast operator to change the type locally. Example:

```
y = cos((double)x);
```

Nondeclaration of Functions

Every function that is called should be declared in the calling function for the types of value it returns. Consider the following program:

```
main()
{
     float a =12.75;
     float b = 7.36;
     printf("%f\n", division(a,b));
}
double division(float x, float y)
{
     return(x/y);
}
```

The function returns a **double** type value but this fact is not known to the calling function and therefore it expects to receive an **int** type value. The program produces either meaningless results or error message such as "redefinition".

The function **division** is like any other variable for the **main** and therefore it should be declared as **double** in the main.

Now, let us assume that the function division is coded as follows:

```
division(float x, float y)
{
   return(x/y);
}
```

Although the values x and y are floats and the result of x/y is also float, the function returns only integer value because no type specifier is given in the function definition. This is wrong too. The function header should include the type specifier to force the function to return a particular type of value.

Missing & Operator in scanf Parameters

All non-pointer variables in a scanf call should be preceded by an **&** operator. If the variable code is declared as an integer, then the statement

```
scanf("%d", code);
```

is wrong. The correct one is scanf("%d", &code);

Remember, the compiler will not detect this error and you may get a crazy output.

Crossing the Bounds of an Array

All C indices start from zero. A common mistake is to start the index from 1. For example, the segment

```
int x[10], sum i;
Sum = 0;
for (i = 1; i < = 10; i++)
     sum = sum + x[i];
```

would not find the correct sum of the elements of array x. The for loop expressions should be corrected as follows:

```
for(i=0;i<10;i++)
```

Forgetting a Space for Null character in a String

All character arrays are terminated with a null character and therefore their size should be declared to hold one character more than the actual string size.

Using Uninitialized Pointers

An uninitialized pointer points to garbage. The following program is wrong.

```
main()
{
        int a, *ptr;
        a = 25;
        *ptr = a+5;
}
```

The pointer **ptr** has not been initialized.

Missing Indirection and Address Operators

Another common error is to forget to use the operators * and & in certain places. Consider the following program:

```
main()
{
        int m, *p1;
```

```
                m = 5;
                pl = m;
                printf("%d\n", *pl);
        }
```

This will print some unknown value because the pointer assignment

```
pl =m;
```

is wrong. It should be:

```
pl = &m;
```

Consider the following expression:

```
y = pl + 10;
```

Perhaps, y was expected to be assigned the value at location **pl** plus 10. But it does not happen. y will contain some unknown address value. The above expression should be rewritten as

```
y = *pl + 10;
```

Missing Parentheses in Pointer Expressions

The following two statements are not the same:

```
x = *pl + 1;
x = *(pl + 1);
```

The first statement would assign the value at location **pl** plus 1 to **x** while the second would assign the value at location **pl + 1**.

Omitting Parentheses around Arguments in Macro Definitions

This would cause incorrect evaluation of expression when the macro definition is substituted.

Example:	**# define** f(x) x * x + 1
The call	**y = f(a+b);**
will be evaluated as	**y = a+b * a+b+1;** which is wrong.

Some other mistakes that we commonly make are:

- Wrong indexing of loops.
- Wrong termination of loops.
- Unending loops.
- Use of incorrect relational test.
- Failure to consider all possible conditions of a variable.
- Trying to divide by zero.
- Mismatching of data specifications and variables in **scanf** and **printf** statements.
- Forgetting truncation and rounding off errors.

15.5 PROGRAM TESTING AND DEBUGGING

Testing and debugging refer to the tasks of detecting and removing errors in a program, so that the program produces the desired results on all occasions. Every programmer should be aware of the fact that rarely does a program run perfectly the first time. No matter how thoroughly the design is carried out, and no matter how much care is taken in coding, one can never say that the program would be 100 per cent error-free. It is therefore necessary to make efforts to detect, isolate and correct any errors that are likely to be present in the program.

Types of Errors

We have discussed a number of common errors. There might be many other errors, some obvious and others not so obvious. All these errors can be classified under four types, namely, syntax errors, run-time errors, logical errors, and latent errors.

Syntax errors: Any violation of rules of the language results in syntax errors. The compiler can detect and isolate such errors. When syntax errors are present, the compilation fails and is terminated after listing the errors and the line numbers in the source program, where the errors have occurred. Remember, in some cases, the line number may not exactly indicate the place of the error. In other cases, one syntax error may result in a long list of errors. Correction of one or two errors at the beginning of the program may eliminate the entire list.

Run-time errors: Errors such as a mismatch of data types or referencing an out-of-range array element go undetected by the compiler. A program with these mistakes will run, but produce erroneous results and therefore the name run-time errors. Isolating a run-time error is usually a difficult task.

Logical errors: As the name implies, these errors are related to the logic of the program execution. Such actions as taking a wrong path, failure to consider a particular condition, and incorrect order of evaluation of statements belong to this category. Logical errors do not show up as compiler-generated error messages. Rather, they cause incorrect results. These errors are primarily due to a poor understanding of the problem, incorrect translation of the algorithm into the program and a lack of clarity of hierarchy of operators. Consider the following statement:

```
if(x ==y)
  printf("They are equal\n");
```

when **x** and **y** are float types values, they rarely become equal, due to truncation errors. The printf call may not be executed at all. A test like **while(x != y)** might create an infinite loop.

Latent errors: It is a 'hidden' error that shows up only when a particular set of data is used. For example, consider the following statement:

```
ratio = (x+y)/(p-q);
```

An error occurs only when p and q are equal. An error of this kind can be detected only by using all possible combinations of test data.

Program Testing

Testing is the process of reviewing and executing a program with the intent of detecting errors, which may belong to any of the four kinds discussed above. We know that while the compiler can detect syntactic and semantic errors, it cannot detect run-time and logical errors that show up during the execution of the program. Testing, therefore, should include necessary steps to detect all possible errors in the program. It is, however, important to remember that it is impractical to find all errors. Testing process may include the following two stages:

1. Human testing.
2. Computer-based testing.

Human testing is an effective error-detection process and is done before the computer-based testing begins. Human testing methods include code inspection by the programmer, code inspection by a test group, and a review by a peer group. The test is carried out statement by statement and is analyzed with respect to a checklist of common programming errors. In addition to finding the errors, the programming style and choice of algorithm are also reviewed.

Computer-based testing involves two stages, namely compiler testing and run-time testing. Compiler testing is the simplest of the two and detects yet undiscovered syntax errors. The program executes when the compiler detects no more errors. Should it mean that the program is correct? Will it produce the expected results? The answer is negative. The program may still contain run-time and logic errors.

Run-time errors may produce run-time error messages such as "null pointer assignment" and "stack overflow". When the program is free from all such errors, it produces output which might or might not be correct. Now comes the crucial test, the test for the *expected output*. The goal is to ensure that the program produces expected results under all conditions of input data.

Test for correct output is done using *test data* with known results for the purpose of comparison. The most important consideration here is the design or invention of effective test data. A useful criteria for test data is that all the various conditions and paths that the processing may take during execution must be tested.

Program testing can be done either at module (function) level or at program level. Module level test, often known as *unit test*, is conducted on each of the modules to uncover errors within the boundary of the module. Unit testing becomes simple when a module is designed to perform only one function.

Once all modules are unit tested, they should be *integrated together* to perform the desired function(s). There are likely to be interfacing problems, such as data mismatch between the modules. An *integration test* is performed to discover errors associated with interfacing.

Program Debugging

Debugging is the process of isolating and correcting the errors. One simple method of debugging is to place print statements throughout the program to display the values of variables. It displays the dynamics of a program and allows us to examine and compare the information at various points. Once the location of an error is identified and the error corrected, the debugging statements may be removed. We can use the conditional compilation statements discussed in Chapter 14 to switch on or off the debugging statements.

Another approach is to use the process of deduction. The location of an error is arrived at using the process of elimination and refinement. This is done using a list of possible causes of the error.

The third error-locating method is to backtrack the incorrect results through the logic of the program until the mistake is located. That is, beginning at the place where the symptom has been uncovered, the program is traced backward until the error is located.