is evaluated every time before execution of the loop body. If this expression is true, the statement in the loop gets executed. In case it is false, the loop terminates and the control of execution is transferred to the statement following the for loop. The third component `i++`, is called *update expression*, and is executed after every execution of the statement in the loop. The fourth component is the loop-body. The program `sumsq1.cpp`, finds the sum and the sum of squares of the first 15 positive even integers.

```
// sumsq1.cpp: sum of first 15 even numbers and their squares' sum
#include<iostream.h>
void main()
{
   int i;
   int sum = 0, sum_of_squares = 0;
   for( i = 2; i <= 30; i += 2 )
   {
      sum += i;
      sum_of_squares += i*i;
   }
   cout << "Sum of first 15 positive even numbers = " << sum << endl;
   cout << "Sum of their squares = " << sum_of_squares;
}
```

Run

```
Sum of first 15 positive even numbers = 240
Sum of their squares = 4960
```

In `main()`, the statement

```
for( i = 2; i <= 30; i += 2 )
```

increments the loop variable `i` by 2 using the update expression

```
i += 2
```

The body of the loop consists of multiple statements, forming a compound statement. The for loop counts from 2 to 30 in steps of two. It is just as easy for a loop to count down, from 30 to 2, as illustrated in the program `sumsq2.cpp`.

```
// sumsq2.cpp: sum of first 15 even numbers and their squares' sum
#include<iostream.h>
void main()
{
   int i;
   int sum = 0, sum_of_squares = 0;
   for( i = 30; i >= 2; i -= 2 )
   {
      sum += i;
      sum_of_squares += i*i;
   }
   cout << "Sum of first 15 positive even numbers = " << sum << endl;
   cout << "Sum of their squares = " << sum_of_squares;
}
```

Run

```
Sum of first 15 positive even numbers = 240
Sum of their squares = 4960
```

Notice the changes: the value of `i` is initialized to 30, the test expression involves the >= condition instead of the <= as in the previous example, and the update expression `i -= 2` decrements the value of `i`. But the output in this case is identical to the first.

The *comma operator* is especially useful in `for` loops. The initialization, test, or update part having multiple expressions can be be separated by commas. For instance,

```
for( i = 0, j=-5; i < 25; i++, j-- )
{
   cout << i << " " << j;
}
```

Another interesting feature of the `for` loop is that any of the three components (the initialization, test and the update components) may be left out, however, the separating semicolons must be present. The variants of the for loop are shown in Figures 5.4.

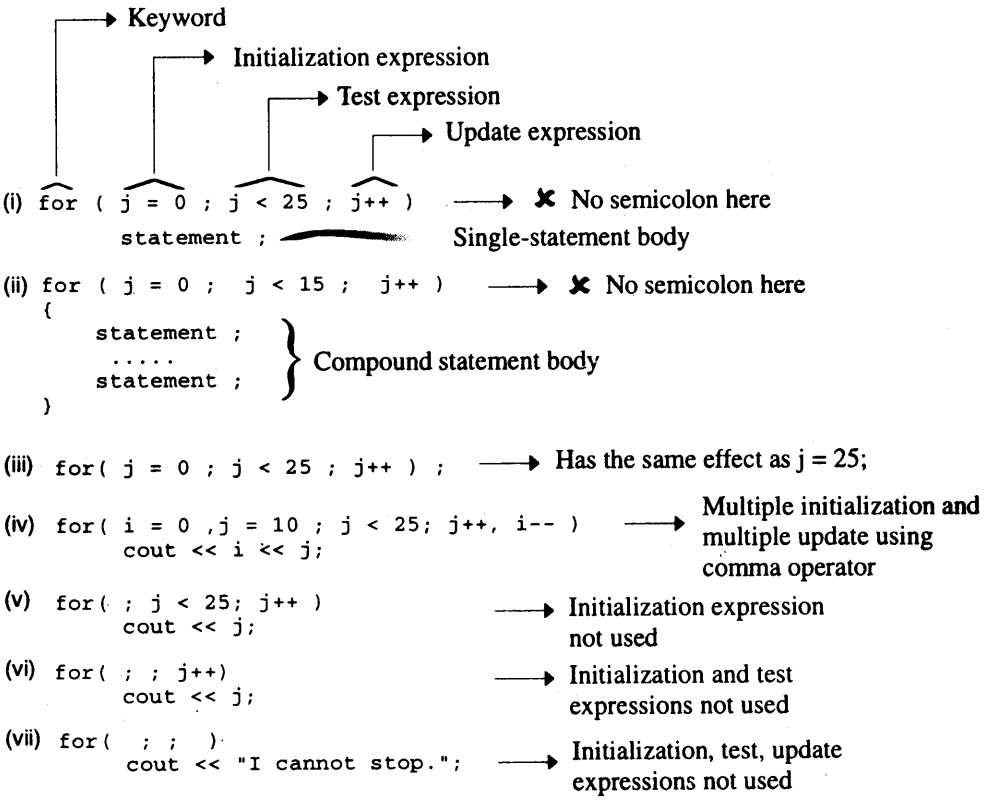


Figure 5.4: Variants of for loop

The program `noinit.cpp`, prints the first 10 multiples of 5, in which the for loop has only the test component.

```
// noinit.cpp: for loop without initialization and updation
#include <iostream.h>
```

```
void main()
{
   int i = 1;
   for( ; i<=10; )
   {
      cout << i*5 << " ";
      ++i;
   }
}
```

Run

```
5 10 15 20 25 30 35 40 45 50
```

In `main()`, the statement

```
int i = 1;
```

is introduced before the for loop. Also, instead of the update expression, `i` is incremented inside the for loop body. Note again that the C++ language does not require the user to indent statements in a for loop. The lines are indented merely for enhancing program appearance (readability).

The nested `for` loops are used extensively in developing programs for solving matrix multiplication, numerical analysis, sorting, and searching problems. The program `pyramid.cpp` illustrates the use of nested `for` loops in generating a pyramid of numbers.

```
// pyramid.cpp: constructs pyramid of digits
#include <iostream.h>
void main()
{
   int p, m, q, n;
   cout << "Enter the number of lines: ";
   cin >> n;
   for(p = 1; p <= n; p++)
   {
      // To print spaces
      for(q = 1; q <= n-p; q++)
         cout << "    ";
      // To print numbers
      m = p;
      for(q = 1; q <= p; q++)
      {
         cout.width(4);
         cout << m++;
      }
      m = m - 2;
      for(q = 1; q < p; q++)
      {
         cout.width(4);
         cout << m--;
      }
      cout << endl;
   }
}
```

Run

```
Enter the number of lines: 5
                1
            2   3   2
        3   4   5   4   3
    4   5   6   7   6   5   4
5   6   7   8   9   8   7   6   5
```

5.7 `while` loop

The `while` loop is used when the number of iterations to be performed are not known in advance. The control flow in the `while` loop is shown in Figure 5.5. The statements in the loop are executed if the test condition is true and the execution continues as long as it remains true. The program `count2.cpp` illustrates the use of the `while` loop to perform the same function as the `for` loop.

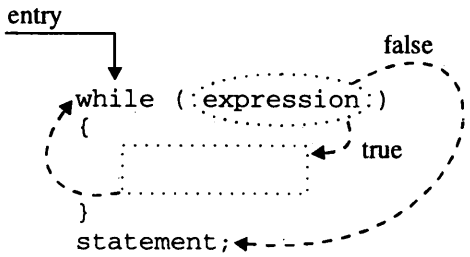


Figure 5.5: Control flow in while loop

```
// count2.cpp: display numbers 1..N using while loop
#include <iostream.h>
void main()
{
   int n;
   cout << "How many integers to be displayed:
   cin >> n;
   int i = 0;
   while( i < n )
   {
      cout << i << endl;
      i++;
   }
}
```

Run

```
How many integers to be displayed: 5
0
1
2
3
4
```

The while loop is often used when the number of times the loop has to be executed is unknown in advance. It is illustrated in the program average1.cpp.

```
// average1.cpp: find the average of the marks
#include <iostream.h>
void main()
{
   int i, sum = 0, count = 0, marks;
   cout << "Enter the marks, -1 at the end...\n";
   cin >> marks;
   while( marks != -1 )
   {
      sum += marks;
      count++;
      cin >> marks;
   }
   float average = sum / count;
   cout << "The average is " << average;
}
```

Run

```
Enter the marks, -1 at the end...
80
75
82
74
-1
The average is 77
```

The first cin statement, just before the while loop, reads the marks scored in the first subject and stores in the variable marks, so that the statement inside the loop can have some valid data to operate. The cin statement inside the loop reads the marks scored in the other subjects one by one. When -1 is entered, the condition

```
marks != -1
```

evaluates to false in the while loop. So, the while loop terminates and the program execution proceeds with the statement immediately after the while loop, which in the above program is

```
average = sum / count;
```

Consider the case when the user inputs -1 as the first marks. The condition in the while statement evaluates to false, and the statements inside the loop are not executed at all. In this case, the value of count continues to be zero, so, while computing the average it leads to division by zero causing a run-time error. This can be prevented by using the if statement as follows:

```
if( count != 0 )
   average = sum / count;
```

The above statement can also be written as

```
if( count )
   average = sum / count;
```

Any expression whose value is nonzero is treated as true. The program binary.cpp illustrates such situations. It uses the while construct to convert a binary number to its decimal equivalent. The shift-left operator << is used for shifting bits stored in a variable in this program.

```
// bin2deci.cpp: conversion of binary number to its decimal equivalent
#include <iostream.h>
void main()
{
   int binary, decimal = 0, digit, position = 0;
   cout << "Enter the binary number: ";
   cin >> binary;
   // converting binary to decimal
   while( binary )
   {
      digit = binary % 10; // extract binary bit
      decimal += digit << position; // newvalue = oldvalue + 2^position
      binary /= 10;  // advance to next bit
      position += 1; // advance to next bit position
   }
   cout << "Its decimal equivalent = " << decimal;
}
```

Run

```
Enter the binary number: 111
Its decimal equivalent = 7
```

5.8 do..while Loop

Sometimes, it is desirable to execute the body of a while loop at least once, even if the test expression evaluates to false during the first iteration. In effect, this requires testing of termination expression at the end of the loop rather than the beginning as in the while loop. So the do-while loop is called a bottom tested loop. The loop is executed as long as the test condition remains true. The control flow in the `do..while` loop is shown in Figure 5.6. Note the semicolon (;) following the while statement at the bottom.

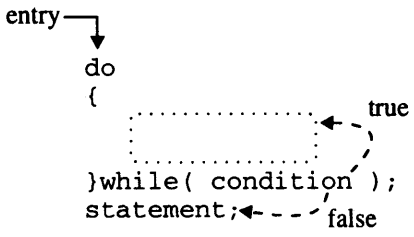


Figure 5.6: Control flow in do..while loop

The program `count3.cpp` illustrates the use of the `do..while` loop.

```
// count3.cpp: display numbers 1..N using do..while loop
#include <iostream.h>
void main()
{
   int n;
   cout << "How many integers to be displayed: ";
   cin >> n;
   int i = 0;
```

```
   do
   {
      cout << i << endl;
      i++;
   } while( i < n );
}
```

Run

```
How many integers to be displayed: 5
0
1
2
3
4
```

To realize the usefulness of the `do..while` construct, consider the following problem: The user has to be prompted to press `m` or `f`. In reality, the user can press any key other than `m` or `f`. In such a case, the message has to be shown again, and the users should be allowed to re-enter one of the two options. An ideal construct to handle such a situation is the `do..while` loop as illustrated in the program `dowhile.cpp`.

```
// dowhile.cpp: do..while loop for asking data until it is valid
#include <iostream.h>
void main()
{
   char inchar;
   do
   {
      cout << "Enter your sex (m/f): ";
      cin >> inchar;
   } while( inchar != 'm' && inchar != 'f' );
   if( inchar == 'm' )
      cout << "So you are male. good!";
   else
      cout << "So you are female. good!";
}
```

Run

```
Enter your sex (m/f): d
Enter your sex (m/f): b
Enter your sex (m/f): m
So you are male. good!
```

In `main()`, the `do..while` loop keeps prompting for the user input until the character `m` for male or `f` for female is entered. Such validation of data is very important while handling sensitive and critical data.

The solution to certain problems inherently requires data validation only after some operation is performed as illustrated in the program `pal.cpp`. It checks if the user entered number is a palindrome using the `do-while` construct.

```
// pal.cpp: to check for a palindrome
#include<iostream.h>
void main()
{
   int n, num, digit, rev = 0;
   cout << "Enter the number: ";
   cin >> num;
   n = num;
   do
   {
      digit = num % 10;
      rev = rev * 10 + digit;
      num /= 10;
   } while( num != 0 );
   cout << "Reverse of the number = " << rev <<  endl;
   if(n == rev)
      cout << "The number is a palindrome\n";
   else
      cout << "The number is not a palindrome\n";
}
```

Run1

```
Enter the number: 123
Reverse of the number = 321
The number is not a palindrome
```

Run2

```
Enter the number: 121
Reverse of the number = 121
The number is a palindrome
```

5.9 `break` Statement

A `break` construct terminates the execution of loop and the control is transferred to the statement immediately following the loop. The term *break* refers to the act of breaking out of a block of code. The control flow in `for`, `while`, and `do-while` loop statements with `break` statement embedded within their body is shown in Figure 5.7.

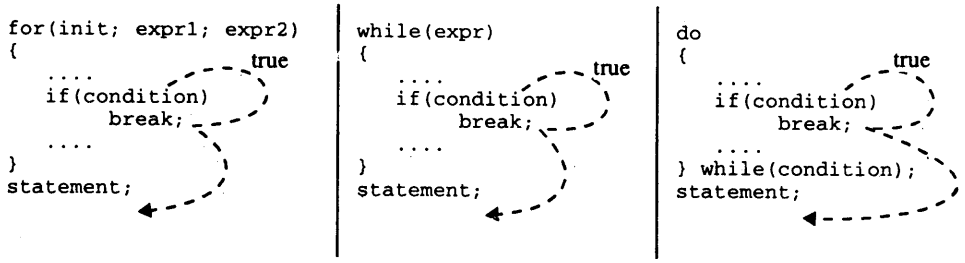


Figure 5.7: break statements in loops

The program average1.cpp discussed earlier has the following code:

```
cin >> marks;
while( marks != -1 )
{
   sum += marks;
   count++;
   cin >> marks;
}
```

It computes the sum of marks entered by the user and maintains their count. This segment of code can be replaced by the following piece of code using the break statement:

```
while( 1 )
{
   cin >> marks;
   if( marks == -1 )
      break;
   sum += marks;
   count++;
}
```

Note that it avoids the use of two cin statements. Whenever -1 is input, the condition marks==-1 evaluates to true, and the break statement is executed, which leads to the termination of loop. Control passes to the statement following the while construct. Observe that the condition in the while loop has been specified as 1 (one) which is nonzero and hence is always true. The condition specifies an infinite loop, but the break prevents such a situation. The above segment of code can also be replaced by the following for loop segment:

```
for( ;; )
{
   cin >> marks;
   if( marks == -1 )
      break;
   sum += marks;
   count++;
}
```

Note that, when test-expression is not mentioned in the for loop, it is implicitly treated as true causing an infinite loop condition. However, it does not lead to an infinite loop as the break statement takes over the responsibility of loop termination. In general, the break statement causes control to pass to the statement following the innermost enclosing for, while, do-while, or switch statement. The same action can also be achieved by using do..while loop as follows:

```
do
{
   cin >> marks;
   if( marks == -1 )
      break;
   sum += marks;
   count++;
} while( 1 );
```

The program average2.cpp illustrates the use of break in loop statements. It performs the same operation as that of the program average1.cpp.

```
// average2.cpp: find the average of the marks
#include <iostream.h>
void main()
{
   int i, sum = 0, count = 0, marks;
   cout << "Enter the marks, -1 at the end...\n";
   while( 1 )
   {
      cin >> marks;
      if( marks == -1 )
         break;
      sum += marks;
      count++;
   }
   float average = sum / count;
   cout << "The average is " << average;
}
```

Run

```
Enter the marks, -1 at the end...
80
75
82
74
-1
The average is 77
```

5.10 `switch` Statement

The `switch` statement provides a clean way to dispatch to different parts of a code based on the value of a single variable or expression. It is a multi-way decision-making construct that allows choosing of a statement (or a group of statements) among several alternatives. The control flow in the switch statement is shown in Figure 5.8. The `switch` statement is mainly used to replace multiple if-else sequence which is hard-to-read and hard-to-maintain.

The expression following the `switch` keyword is an integer valued expression. The value of this expression decides the sequence of statements to be executed. Each sequence of statements begins with the keyword `case` followed by a constant integer. (Note that constant characters may also be specified). Control is transferred to the statements following the case label whose constant is equal to the value of the expression in the `switch` statement. The `default` part is optional in the `switch` statement. The keyword `break` is used to delimit the scope of the statements under a particular case.

```
switch( option )
{
   case 1: cout << "Option # 1 entered";
      break;
   case 2: cout << "Option # 2 entered";
      break;
   default: cout << "Invalid option entered";
}
```

In the above segment, if `option` is 1, then the first `cout` will be executed and the control will pass to the next statement after the `switch`. Otherwise, the rest of the case statement will be evaluated in the same way. If none of them match, then the last `cout` with the `default` will be executed.

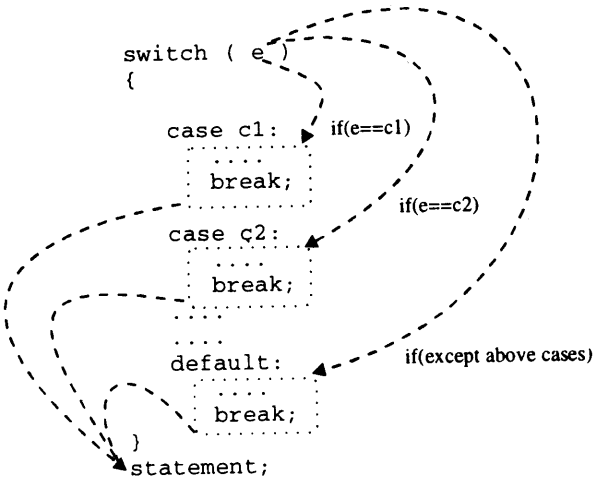


Figure 5.8: Control flow in switch statement

The `break` statement is essential for the correct realization of the `switch` structure. It causes exit from the switch structure after the case statements are executed. The `break` can be omitted in which case the control falls through to the next `case` statements. For example, omitting the `break` statement in the first `case` statement will cause both the `case 1` and `case 2`'s body to be executed. The `break` statements can be omitted when the same operation is to be performed for a number of cases as illustrated below:

```
switch( ch )
{
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u': ++ vowel;
      break;
   case ' ': ++ spaces;
     break;
   default : ++ consonant;
}
```

In the above segment, when the contents of `ch` is equal to a vowel character, the statement

```
++vowel;
```

is executed.

The different cases and the `default` keyword may appear in any order. The program `sex2.cpp` illustrates the use of `switch` construct in replacing the nested `if-else` statements.

```
// sex2.cpp: use of switch statement
#include <iostream.h>
void main()
{
   char ch;
   cout << "Enter your sex (m/f): ";
   cin >> ch;
   switch( ch )
   {
      case 'm':
         cout << "So you are male. good!";
         break;
      case 'f':
         cout << "So you are female. good!";
         break;
      default:    // if none of the above match any cases
         cout << "Error: Invalid sex code!";
   }
}
```

Run1

```
Enter your sex (m/f): m
So you are male. good!
```

Run2

```
Enter your sex (m/f): b
Error: Invalid sex code!
```

5.11 `continue` Statement

The `continue` statement skips the remainder of the current iteration and initiates the execution of the next iteration. When this statement is encountered in a loop, the rest of the statements in the loop are skipped, and the control passes to the condition, which is evaluated, and if true, the loop is entered again. The `continue` statement has the following syntax:

```
continue;
```

The control flow in `for`, `while`, and `do..while` loops with `continue` statement embedded within their body is shown in Figure 5.9.

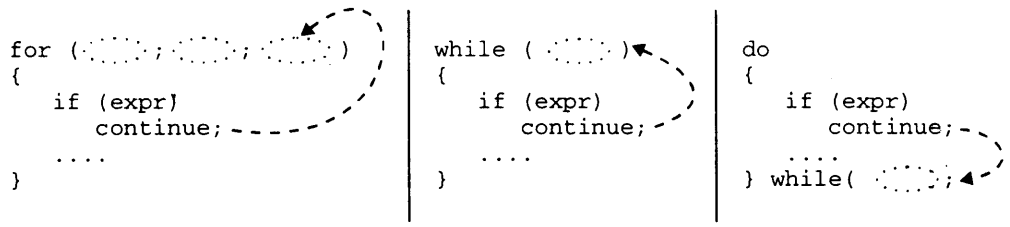


Figure 5.9: Operational flow with continue statement

The program `sumpos.cpp` accepts an indefinite number of values from the keyboard and prints the sum of only the positive numbers. It demonstrates the use of `break` and `continue` statements.

```
// sumpos.cpp: sum of positive numbers
#include <iostream.h>
void main()
{
   int num, total = 0;
   do
   {
      cout << "Enter a number (0 to quit): ";
      cin >> num;
      if( num == 0 )
      {
         cout << "end of data entry." << endl;
         break;       // terminates loop
      }
      if( num < 0 )
      {
         cout << "skipping this number." << endl;
         continue; // skips next statements and transfers to start of loop
      }
      total += num;
   } while(1);
   cout << "Total of all +ve numbers is " << total;
}
```

Run

```
Enter a number (0 to quit): 10
Enter a number (0 to quit): 20
Enter a number (0 to quit): -5
skipping this number.
Enter a number (0 to quit): 10
Enter a number (0 to quit): 0
end of data entry.
Total of all +ve numbers is 40
```

In `do..while` loop of the above program, on encountering `break`, control is transferred outside the loop. On encountering `continue`, control is transferred to the `while` condition which is always true (nonzero). Figure 5.10 shows action differences between `break` and `continue` statements in loops. The `break` and `continue` statements must be judiciously used and their indiscriminate use can hamper the clarity of the logic.

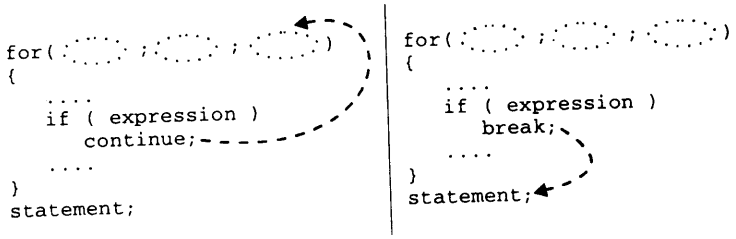


Figure 5.10: Control flow for continue and break

5.12 goto Statement

The C++ language also provides the much abused `goto` statement for branching unconditionally to any part of a program . A debate on whether the use of the `goto` construct in structured programming is essential or not, is purely academic, but practically, the `goto`, is never necessary and therefore is not used by many programmers. However, there are certain places where the use of `goto` becomes mandatory. For instance, to exit from some deeply nested loops, `goto` can be used. The general format of a `goto` statement is:

```
goto label;
```

Here `label` is an identifier used to label the target statement to which the control should be transferred. Control may be transferred to any other statement within the current function. The target statement must be labeled and the `label` must be followed by a colon. The target statement will appear as

```
label: statement;
```

Note that the declaration of the `label` symbol is not required. The program `jump.cpp` is equivalent to the program `sumpos.cpp` discussed above. It uses `goto` statement instead of the `break` statement.

```
// jump.cpp: sum of positive numbers using goto construct
#include <iostream.h>
void main()
{
   int num, total = 0;
   do
   {
      cout << "Enter a number (0 to quit): ";
      cin >> num;
      if( num == 0 )
      {
         cout << "end of data entry." << endl;
         goto dataend;     // transfer to dataend position
      }
      if( num < 0 )
      {
         cout << "skipping this number." << endl;
         continue; // skips next statements and transfers to start of loop
      }
      total += num;
   } while(1);
   dataend: cout << "Total of all +ve numbers is " << total;
}
```

Run

```
Enter a number (0 to quit): 10
Enter a number (0 to quit): 20
Enter a number (0 to quit): -5
skipping this number.
Enter a number (0 to quit): 10
Enter a number (0 to quit): 0
end of data entry.
Total of all +ve numbers is 40
```

Any loop (`for`, `while`, or `do..while`) statement can be replaced by an if statement coupled with a `goto` statement. But this, of course makes the program unreadable. On the other hand, there are situations wherein `goto` statement can make the flow of control more obvious. For example, the following segment determines whether two arrays `x` and `y` have an element in common or not. The element `x` has `n` elements and `y` has `m` elements.

```
for( i = 0; i < n; i++ )
   for( j = 0; j < m; j++ )
      if( x[ i ] == y[ j ] )
          goto found;
// Element not found
...
found:
// Element found
...
```

Except in cases such as the one cited above, the use of the `goto` statement must be avoided.

It is possible to use `goto` statement to jump from outside a loop to inside the loop body, but it is logically incorrect. Hence, goto jumps shown in Figure 5.11 would cause problems and therefore must be avoided.

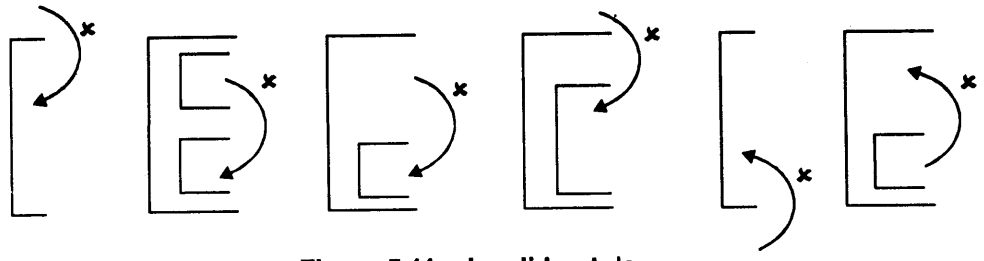


Figure 5.11: Invalid goto's

5.13 Wild Statements

It is very difficult to detect semantic errors in a program when semicolons are used improperly with loops. One such case is illustrated in the program `age5.cpp`

```
// age5.cpp: if statement with wrong usage of syntax
#include <iostream.h>
void main()
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if( age > 12 && age < 20 );
      cout << "you are a teen-aged person. good!";
}
```

Run1

```
Enter your age: 14
you are a teen-aged person. good!
```

Run2

```
Enter your age: 50
you are a teen-aged person. good!
```

In `main()`, the statement

```
if( age > 12 && age < 20 );
```

effectively does nothing; observe the semicolon after the condition statement. The program displays the same message for any type of input data. Whether the input age lies in range of teenage or not, it produces the message

```
you are a teen-aged person. good!
```

See *Run2* output which shows even 50 year aged person as teen-aged!

Equality Test

The program `agecmp.cpp` is written for comparing ages of two persons. It prints the illogical message except for some typical value.

```
// agecmp.cpp: age comparison
#include <iostream.h>
void main()
{
   int myage = 25, yourage;
   cout << "Hi! my age is " << myage << endl;
   cout << "What is your age ? ";
   cin >> yourage;
   if( myage = yourage )
      cout << "We are born on the same day. Are we twins!";
}
```

Run1

```
Hi! my age is 25
What is your age ? 25
We are born on the same day. Are we twins!
```

Run2

```
Hi! my age is 25
What is your age ? 10
We are born on the same day. Are we twins!
```

Run3

```
Hi! my age is 25
What is your age ? 0
```

The statement in `main()`

```
if( myage = yourage )
```

has the expression `myage = yourage`. It assigns the contents of the variable `yourage`, to `myage`. It is evaluated to true, for all nonzero values of `yourage` and hence, the program prints the same message except for zero input value. The programmer must be careful while writing the statement, which checks for the equality of data.

Review Questions

5.1 Discuss the need of control flow statements in C++.

5.2 What are the differences between break and continue statements ? Develop an interactive program which illustrates the differences.

5.3 Justify that "goto statement cannot be used to transfer control from outside to inside the loop"

5.4 Write an interactive program to print a given integer in the reverse order. For instance, 1234 should be printed as 4321.

5.5 Write an optimized algorithm (program) to print the first N prime numbers, where N is a number accepted from the keyboard.

5.6 Write a program to print the sum of all squares between 1 and N, where N is a number accepted from the keyboard. i.e., `1 + 4 + .... + (N*N)`.

5.7 Develop a program to find the roots of a quadratic equation. Use switch statements to handle different values of the discriminant (b^2-4*a*c).

5.8 State which of the following statements are TRUE or FALSE. Give reasons.

(a) Use of goto helps in developing structured programming.

(b) In `if` statement, if the if condition fails, else-part is executed.

(c) The value -1 is treated as false.

(d) The `switch` statement can have more than one matching cases.

(e) The `break` statement terminates the execution of the loop.

(f) Explicit transfer of control from outside the loop to inside is logically correct.

(g) The use of an expression such as a = b as a test expression is not encouraged.

5.9 Write a program to compute the exponential value of a given number x using the series:

$e(x) = 1+x+x^2/2!+x^3/3!+...$

5.10 Write an interactive program for computing the *factorial* of a number using the while loop.

5.11 Write a program to generate reverse pyramid of digits.

5.12 Write an interactive program to compute the *cosine* of a number using the series:

$\cos(x)=1-x^2/2!+x^4/4!-x^6/6!+...$

5.13 Write an interactive program to compute the area of a triangle for the following cases:

a) for 3 sides of a triangle (a, b, and c):

```
p = a + b + c;
s = (a + b + c)/2;
area = sqrt((double)(s*(s-a)*(s-b)*(s-c)));
```

b) for right angle triangle: `area = (base*height)/2;`

5.14 Write a program to print the multiplication table using do..while loop.

5.15 Write an interactive program to draw a histogram of marks scored in different subjects as follows:

```
subject1: ********************** (50%)
subject2: ********************************** (72%)
```

5.16 Write a program to print a conversion chart of various currencies as shown in the table below:

```
-------------------------------------------
 US $      Rs      Dinar      Yen      Pound
-------------------------------------------
 ....      ....     ....       ....     ....
-------------------------------------------
```

6

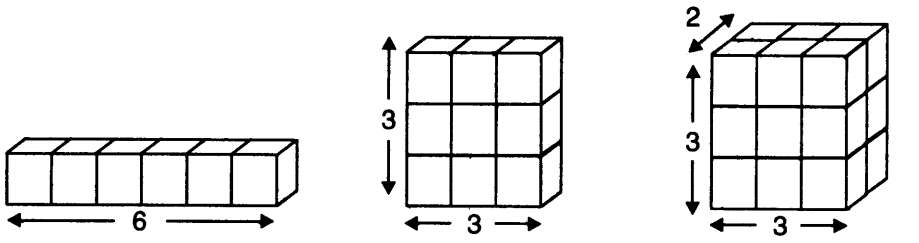
Arrays and Strings

6.1 Introduction

An array is a group of logically related data items of the same data-type addressed by a common name, and all the items are stored in contiguous (physically adjacent) memory locations. For instance, the statement

```
int marks[10];
```

defines an array by the name `marks` that can hold a maximum of ten elements. The individual elements of an array are accessed and manipulated using the array name followed by their index. The marks scored in the first subject is accessed as `marks[0]` and the marks scored in the 10th subject as `marks[9]`. In this case, a sequence of ten integers representing the marks are stored one after another in memory. A sequence of characters is called *string*. It can be used for storing and manipulating text such as words, names, and sentences. The arrays can be used to represent a vector, matrix, etc., as shown in Figure 6.1.



Vector Matrix Three dimensional array

Figure 6.1: Single and multidimensional arrays

6.2 Operations on Arrays

To see the usefulness of arrays, consider the problem of reading the ages of five persons and computing the average age. Five variables need to be defined for storing the age of five persons and they have to be read and processed using distinct statements as illustrated in the program `age1.cpp`.

```
// age1.cpp: multiple variables to handle data which are logically same
#include <iostream.h>
void main()
{
   int age1, age2, age3, age4, age5;
   float sum = 0;
```

```
   cout << "Enter person 1 age: ";
   cin >> age1;
   sum += age1;
   cout << "Enter person 2 age: ";
   cin >> age2;
   sum += age2;
   cout << "Enter person 3 age: ";
   cin >> age3;
   sum += age3;
   cout << "Enter person 4 age: ";
   cin >> age4;
   sum += age4;
   cout << "Enter person 5 age: ";
   cin >> age5;
   sum += age5;
   cout << "Average age = " << sum/5;
}
```

Run

```
Enter person 1 age: 23
Enter person 2 age: 40
Enter person 3 age: 30
Enter person 4 age: 27
Enter person 5 age: 25
Average age = 29
```

The above program uses distinct statements to read and add the age of each person. The resulting value of summation is stored in the variable sum. Finally, the average age is computed by dividing the sum by 5. A program written in this style is very clumsy, and difficult to enhance. If there are a large number of individuals, the number of statements increase proportionately. A more elegant approach is to use an array type variable to store the age of persons, and process them using loops as illustrated in the program age2.cpp.

```
// age2.cpp: arrays to handle data which are of the same type
#include <iostream.h>
void main()
{
   int age[5];          // array definition
   float sum = 0;
   for( int i = 0; i < 5; i++ )
   {
      cout << "Enter person " << i+1 << " age: ";
      cin >> age[i];    // reading array elements
   }
   for( i = 0; i < 5; i++ )
      sum += age[i];    // array manipulation
   cout << "Average age = " << sum/5;
}
```

Run

```
Enter person 1 age: 23
```

```
Enter person 2 age: 40
Enter person 3 age: 30
Enter person 4 age: 27
Enter person 5 age: 25
Average age = 29
```

Handling arrays involve array definition, array initialization, and accessing elements of an array. In `main()`, the statement

```
int age[5];
```

defines an array of five elements of integer type with the name `age`. It reserves `5*sizeof(int)` bytes of memory space for storing the five integer numbers. The statement

```
cin >> age[i];
```

reads each integer value and stores it in the array element indexed by the variable `i`. Here, the variable `i` is known as the *array index* or *subscript* and hence, arrays are popularly called *subscripted variables*. Note that an array of `N` elements has indexes in the range 0 to N-1. The statement

```
sum += age[i];
```

accesses the contents of the $(i+1)^{th}$ element of the array `age` and adds it to the variable `sum`.

Array Definition

Like other normal variables, the array variable must be defined before its use. The syntax for defining an array is shown in Figure 6.2.

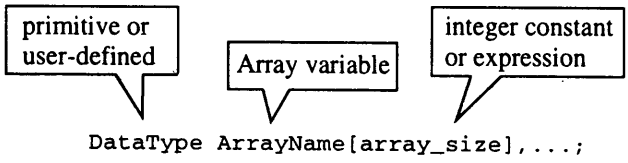


Figure 6.2: Array definition

In the definition, the array name must be a valid C++ variable, followed by an integer value enclosed in square braces. The integer value indicates the maximum number of elements the array can hold. The following are some valid array definition statements:

```
int marks[100];             // integer array of size 100
float salary[25];           // floating-point array of size 25
char name[50];              // character array of size 50
int a[10], b[12], c[25];    // defines three arrays
double d1, num[10];         // defines a variable and double array
```

The last statement indicates that a normal variable and array can be defined in a single statement. The representation of an array defined using the statement

```
int age[5];
```

is shown in Figure 6.3 by assuming that each element of the array (i.e., each integer) occupies two bytes.

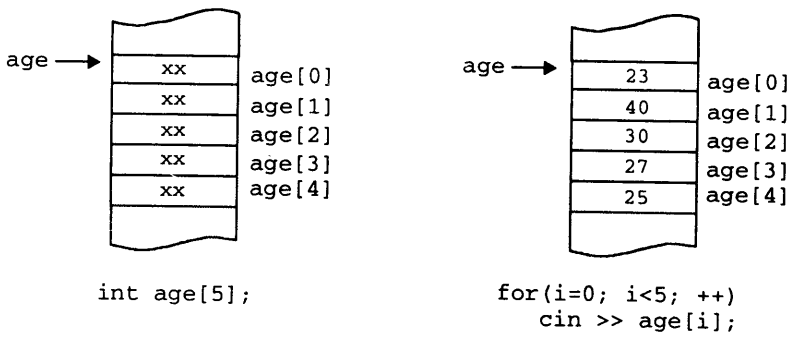


Figure 6.3: Storage representation for an array

Accessing Array Elements

Once an array variable is defined, its elements can be accessed by using an index. The syntax for accessing array elements is shown in Figure 6.4.

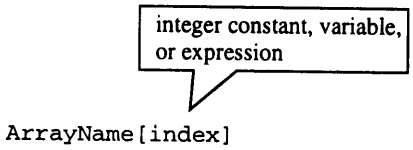


Figure 6.4: Accessing an array element

To access a particular element in the array, specify the array name followed by an integer constant or variable (array index) enclosed within square braces. The array index, indicates the element of the array, which has to be accessed. For instance, the expression

```
age[4]
```

accesses the 5th element of the array `age`. Note that, in an array of N elements, the first element is indexed by zero and the last element of an array is indexed by N-1. The loop used to read the elements of the array is:

```
for( int i = 0; i < 5; i++ )
{
   cout << "Enter person " << i+1 << " age: ";
   cin >> age[i];
}
```

The variable `i` varies from 0 to N-1 (i.e., 0 to 4 in the above segment). Statements such as,

```
age[i]++;
```

can be used to increment the value of the `i`th item in the array `age` and hence the following,

```
age[i] = 11;
age[3] = 25;
```

are valid statements. Note that, *the expression* `age[i]` *can also be represented as* `i[age]`*; similarly, the expression* `age[3]` *is equivalent to* `3[age]`.

The program `nodup.cpp` illustrates the manipulation of a vector. It reads a vector and removes all duplicate elements in that vector. The vector is adjusted after removing all the duplicate elements.