C++ How To Program by Deitel & Deitel (2nd Edition) Pdf Download

Computer Science

University of Sargodha

363 pag.

C++ HOW TO PROGRAM

SECOND EDITION

Illustrations List *(Main Page)*

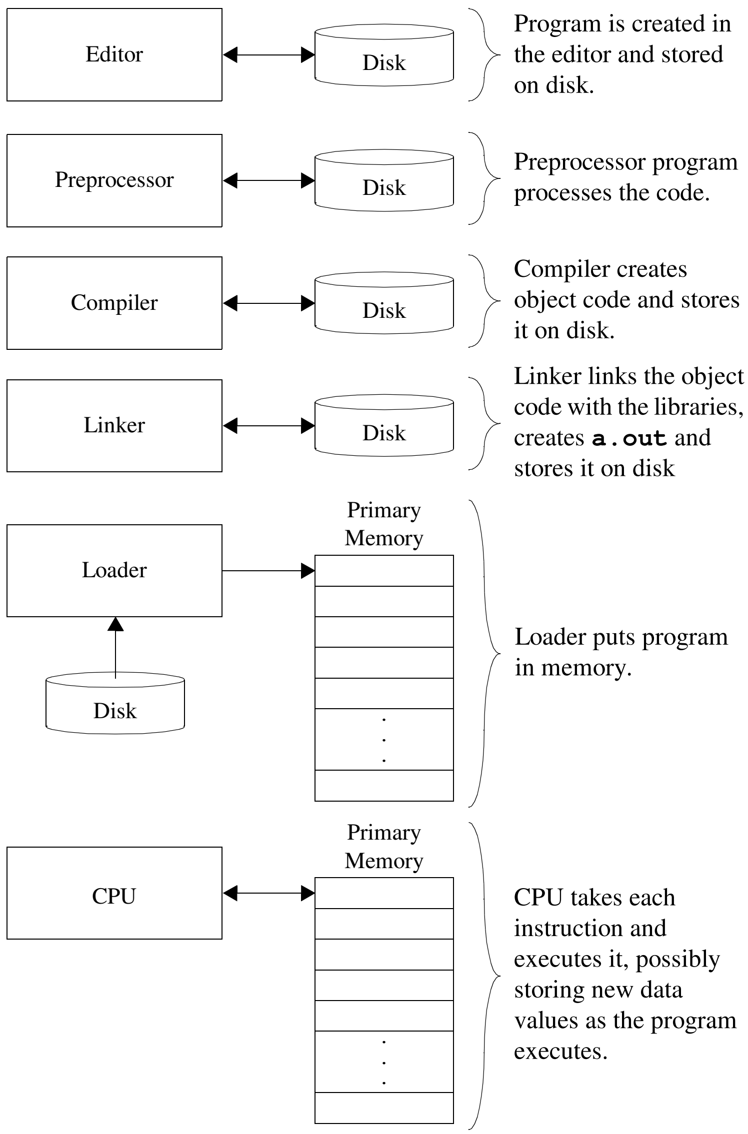


Fig. 1.1 A typical C++ environment.

```
// Fig. 1.2: fig01_02.cpp
// A first program in C++
#include <iostream.h>

int main()
{
   cout << "Welcome to C++!\n";

   return 0;   // indicate that program ended successfully
}
```

```
Welcome to C++!
```

Fig. 1.2 Text printing program.

Escape Sequence	Description
`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Used to print a backslash character.
`\"`	Double quote. Used to print a double quote character.

Fig. 1.3 Some common escape sequences.

```
// Fig. 1.4: fig01_04.cpp
// Printing a line with multiple statements
#include <iostream.h>

int main()
{
   cout << "Welcome ";
   cout << "to C++!\n";

   return 0;   // indicate that program ended successfully
}
```

```
Welcome to C++!
```

Fig. 1.4 Printing on one line with separate statements using `cout`.

```
// Fig. 1.5: fig01_05.cpp
// Printing multiple lines with a single statement
#include <iostream.h>

int main()
{
   cout << "Welcome\nto\n\nC++!\n";

   return 0;   // indicate that program ended successfully
}
```

```
Welcome
to

C++!
```

Fig. 1.5 Printing on multiple lines with a single statement using **cout**.

```
// Fig. 1.6: fig01_06.cpp
// Addition program
#include <iostream.h>

int main()
{
   int integer1, integer2, sum;      // declaration

   cout << "Enter first integer\n";  // prompt
   cin >> integer1;                  // read an integer
   cout << "Enter second integer\n"; // prompt
   cin >> integer2;                  // read an integer
   sum = integer1 + integer2;        // assignment of sum
   cout << "Sum is " << sum << endl; // print sum

   return 0;   // indicate that program ended successfully
}
```

Fig. 1.6 An addition program (part 1 of 2).

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Fig. 1.6 An addition program (part 2 of 2).

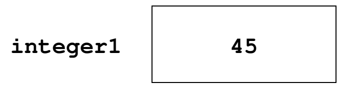


Fig. 1.7 A memory location showing the name and value of a variable.

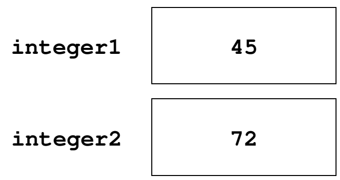


Fig. 1.8 Memory locations after values for two variables have been input.

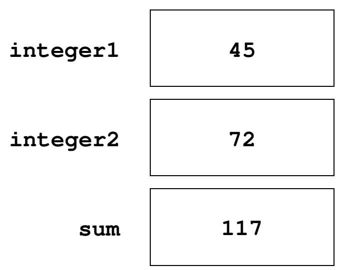


Fig. 1.9 Memory locations after a calculation.

C++ operation	Arithmetic operator	Algebraic expression	C++ expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	`%`	$r \bmod s$	`r % s`

Fig. 1.10 Arithmetic operators.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`*`, `/`, or `%`	Multiplication Division Modulus	Evaluated second. If there are several, they are evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Fig. 1.11 Precedence of arithmetic operators.

Step 1. `y = 2 * 5 * 5 + 3 * 5 + 7;`

`2 * 5 is` $\boxed{10}$ (Leftmost multiplication)

Step 2. `y = 10 * 5 + 3 * 5 + 7;`

`10 * 5 is` $\boxed{50}$ (Leftmost multiplication)

Step 3. `y = 50 + 3 * 5 + 7;`

`3 * 5 is` $\boxed{15}$ (Multiplication before addition)

Step 4. `y = 50 + 15 + 7;`

`50 + 15 is` $\boxed{65}$ (Leftmost addition)

Step 5. `y = 65 + 7;`

`65 + 7 is` $\boxed{72}$ (Last addition)

Step 6. `y = 72;` (Last operation—place **72** into **y**)

Fig. 1.12 Order in which a second-degree polynomial is evaluated.

Standard algebraic equality operator or relational operator	C++ equality or relational operator	Example of C++ condition	Meaning of C++ condition
Equality operators			
=	`==`	`x == y`	**x** is equal to **y**
≠	`!=`	`x != y`	**x** is not equal to **y**
Relational operators			
>	`>`	`x > y`	**x** is greater than **y**
<	`<`	`x < y`	**x** is less than **y**
≥	`>=`	`x >= y`	**x** is greater than or equal to **y**
≤	`<=`	`x <= y`	**x** is less than or equal to **y**

Fig. 1.13 Equality and relational operators.

```
// Fig. 1.14: fig01_14.cpp
// Using if statements, relational
// operators, and equality operators
#include <iostream.h>

int main()
{
   int num1, num2;

   cout << "Enter two integers, and I will tell you\n"
        << "the relationships they satisfy: ";
   cin >> num1 >> num2;   // read two integers

   if ( num1 == num2 )
      cout << num1 << " is equal to " << num2 << endl;

   if ( num1 != num2 )
      cout << num1 << " is not equal to " << num2 << endl;

   if ( num1 < num2 )
      cout << num1 << " is less than " << num2 << endl;

   if ( num1 > num2 )
      cout << num1 << " is greater than " << num2 << endl;

   if ( num1 <= num2 )
      cout << num1 << " is less than or equal to "
           << num2 << endl;

   if ( num1 >= num2 )
      cout << num1 << " is greater than or equal to "
           << num2 << endl;

   return 0;   // indicate that program ended successfully
}
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

Fig. 1.14 Using equality and relational operators (part 1 of 2).

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Fig. 1.14 Using equality and relational operators (part 2 of 2).

Operators	Associativity	Type
`()`	left to right	parentheses
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`<< >>`	left to right	stream insertion/extraction
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`=`	right to left	assignment

Fig. 1.15 Precedence and associativity of the operators discussed so far.

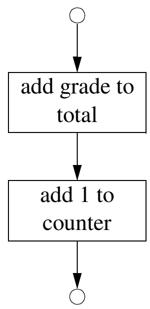
1

```
// Fig. 1.16: fig01_16.cpp
// Using new-style header files
#include <iostream>

using namespace std;

int main()
{
   cout << "Welcome to C++!\n";
   std::cout << "Welcome to C++!\n";

   return 0;   // indicate that program ended successfully
}
```

```
Welcome to C++!
Welcome to C++!
```

Fig. 1.16 Using new-style header files.

Illustrations List *(Main Page)*




```
total = total + grade;
counter = counter + 1;
```

Fig. 2.1 Flowcharting C++'s sequence structure.

C++ Keywords

C and C++ keywords				
`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			
C++ only keywords				
`asm`	`bool`	`catch`	`class`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`false`	`friend`
`inline`	`mutable`	`namespace`	`new`	`operator`
`private`	`protected`	`public`	`reinterpret_cast`	
`static_cast`	`template`	`this`	`throw`	`true`
`try`	`typeid`	`typename`	`using`	`virtual`
`wchar_t`				

Fig. 2.2 C++ keywords.

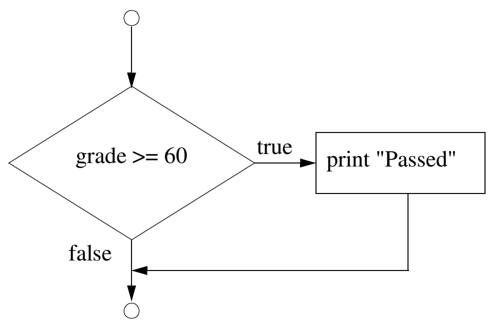


Fig. 2.3 Flowcharting the single-selection `if` structure.

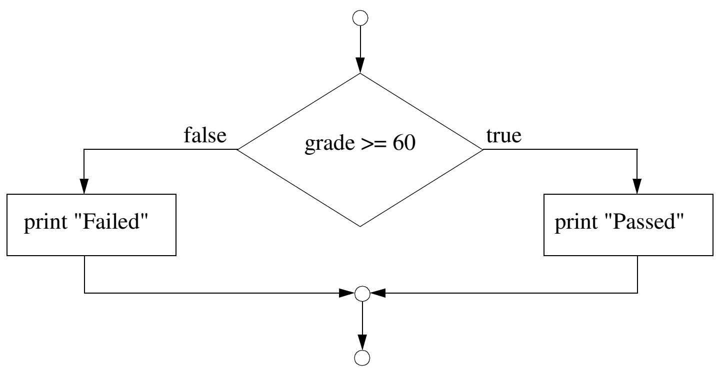


Fig. 2.4 Flowcharting the double-selection `if/else` structure.

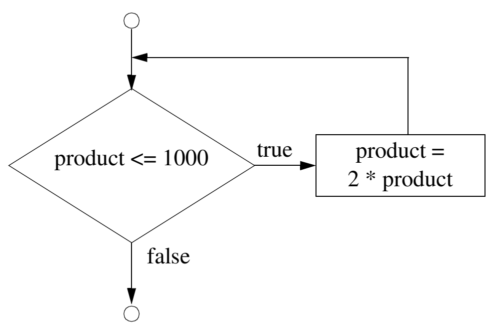


Fig. 2.5 Flowcharting the `while` repetition structure.

Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

Fig. 2.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

```
// Fig. 2.7: fig02_07.cpp
// Class average program with counter-controlled repetition
#include <iostream.h>

int main()
{
   int total,        // sum of grades
       gradeCounter, // number of grades entered
       grade,        // one grade
       average;      // average of grades

   // initialization phase
   total = 0;                          // clear total
   gradeCounter = 1;                   // prepare to loop

   // processing phase
   while ( gradeCounter <= 10 ) {      // loop 10 times
      cout << "Enter grade: ";         // prompt for input
      cin >> grade;                    // input grade
      total = total + grade;           // add grade to total
      gradeCounter = gradeCounter + 1; // increment counter
   }

   // termination phase
   average = total / 10;               // integer division
   cout << "Class average is " << average << endl;

   return 0;   // indicate program ended successfully
}
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 2.7 C++ program and sample execution for the class average problem with counter-controlled repetition.

Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"

Fig. 2.8 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class average problem.

```
// Fig. 2.9: fig02_09.cpp
// Class average program with sentinel-controlled repetition.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int total,          // sum of grades
       gradeCounter,   // number of grades entered
       grade;          // one grade
   float average;      // number with decimal point for average

```

Fig. 2.9 C++ program and sample execution for the class average problem with sentinel-controlled repetition (part 1 of 2).

```
   // initialization phase
   total = 0;
   gradeCounter = 0;

   // processing phase
   cout << "Enter grade, -1 to end: ";
   cin >> grade;

   while ( grade != -1 ) {
      total = total + grade;
      gradeCounter = gradeCounter + 1;
      cout << "Enter grade, -1 to end: ";
      cin >> grade;
   }

   // termination phase
   if ( gradeCounter != 0 ) {
      average = static_cast< float >( total ) / gradeCounter;
      cout << "Class average is " << setprecision( 2 )
           << setiosflags( ios::fixed | ios::showpoint )
           << average << endl;
   }
   else
      cout << "No grades were entered" << endl;

   return 0;   // indicate program ended successfully
}
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Fig. 2.9 C++ program and sample execution for the class average problem with sentinel-controlled repetition (part 2 of 2).

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"
```

Fig. 2.10 Pseudocode for examination results problem.

```cpp
// Fig. 2.11: fig02_11.cpp
// Analysis of examination results
#include <iostream.h>

int main()
{
   // initialize variables in declarations
   int passes = 0,           // number of passes
       failures = 0,         // number of failures
       studentCounter = 1,   // student counter
       result;               // one exam result

   // process 10 students; counter-controlled loop
   while ( studentCounter <= 10 ) {
      cout << "Enter result (1=pass,2=fail): ";
      cin >> result;

```

Fig. 2.11 C++ program and sample executions for examination results problem (part 1 of 2).

```cpp
      if ( result == 1 )        // if/else nested in while
         passes = passes + 1;
      else
         failures = failures + 1;

      studentCounter = studentCounter + 1;
   }

   // termination phase
   cout << "Passed " << passes << endl;
   cout << "Failed " << failures << endl;

   if ( passes > 8 )
      cout << "Raise tuition " << endl;

```

```
    return 0;    // successful termination
}
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Fig. 2.11 C++ program and sample executions for examination results problem (part 2 of 2).

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 2.12 Arithmetic assignment operators.

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment **a** by 1, then use the new value of **a** in the expression in which **a** resides.
`++`	postincrement	`a++`	Use the current value of **a** in the expression in which **a** resides, then increment **a** by 1.
`--`	predecrement	`--b`	Decrement **b** by 1, then use the new value of **b** in the expression in which **b** resides.
`--`	postdecrement	`b--`	Use the current value of **b** in the expression in which **b** resides, then decrement **b** by 1.

Fig. 2.13 The increment and decrement operators.

```
// Fig. 2.14: fig02_14.cpp
// Preincrementing and postincrementing
#include <iostream.h>

int main()
{
   int c;

   c = 5;
   cout << c << endl;         // print 5
   cout << c++ << endl;       // print 5 then postincrement
   cout << c << endl << endl; // print 6

   c = 5;
   cout << c << endl;         // print 5
   cout << ++c << endl;       // preincrement then print 6
   cout << c << endl;         // print 6

   return 0;                  // successful termination
}
```

```
5
5
6

5
6
6
```

Fig. 2.14 The difference between preincrementing and postincrementing.

Operators	Associativity	Type
()	left to right	parentheses
++ -- + - static_cast<*type*>()	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 2.15 Precedence of the operators encountered so far in the text.

```
// Fig. 2.16: fig02_16.cpp
// Counter-controlled repetition
#include <iostream.h>

int main()
{
   int counter = 1;              // initialization

   while ( counter <= 10 ) {     // repetition condition
      cout << counter << endl;
      ++counter;                 // increment
   }

   return 0;
}
```

Fig. 2.16 Counter-controlled repetition.

```
1
2
3
4
5
6
7
8
9
10
```

```
// Fig. 2.17: fig02_17.cpp
// Counter-controlled repetition with the for structure
#include <iostream.h>

int main()
{
   // Initialization, repetition condition, and incrementing
   // are all included in the for structure header.

   for ( int counter = 1; counter <= 10; counter++ )
      cout << counter << endl;

   return 0;
}
```

Fig. 2.17 Counter-controlled repetition with the `for` structure.

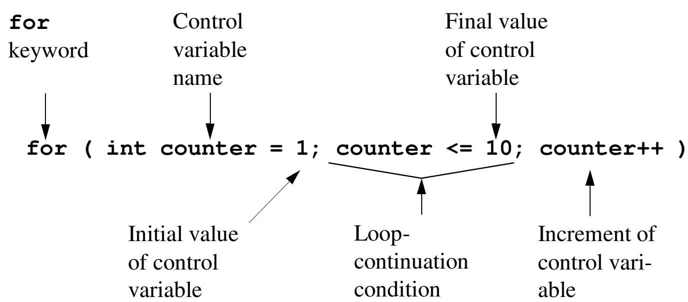


Fig. 2.18 Components of a typical `for` header.

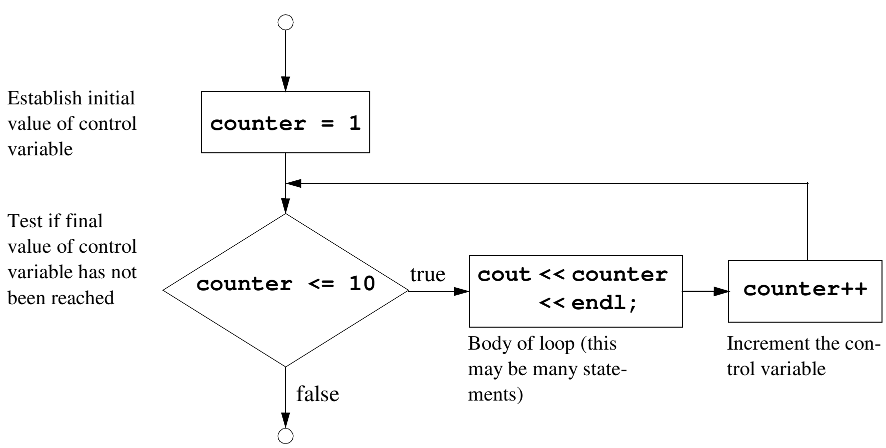


Fig. 2.19 Flowcharting a typical `for` repetition structure.

```
// Fig. 2.20: fig02_20.cpp
// Summation with for
#include <iostream.h>

int main()
{
   int sum = 0;

   for ( int number = 2; number <= 100; number += 2 )
      sum += number;

   cout << "Sum is " << sum << endl;

   return 0;
}
```

```
Sum is 2550
```

Fig. 2.20 Summation with **`for`**.

```
// Fig. 2.21: fig02_21.cpp
// Calculating compound interest
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
   double amount,              // amount on deposit
          principal = 1000.0,  // starting principal
          rate = .05;          // interest rate

   cout << "Year" << setw( 21 )
        << "Amount on deposit" << endl;

   for ( int year = 1; year <= 10; year++ ) {
      amount = principal * pow( 1.0 + rate, year );
      cout << setw( 4 ) << year
           << setiosflags( ios::fixed | ios::showpoint )
           << setw( 21 ) << setprecision( 2 )
           << amount << endl;
   }

   return 0;
}
```

Fig. 2.21 Calculating compound interest with **`for`** (part 1 of 2).

```
Year    Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.62
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Fig. 2.21 Calculating compound interest with **for** (part 2 of 2).

```
// Fig. 2.22: fig02_22.cpp
// Counting letter grades
#include <iostream.h>

int main()
{
   int grade,       // one grade
       aCount = 0,  // number of A's
       bCount = 0,  // number of B's
       cCount = 0,  // number of C's
       dCount = 0,  // number of D's
       fCount = 0;  // number of F's

   cout << "Enter the letter grades." << endl
        << "Enter the EOF character to end input." << endl;

   while ( ( grade = cin.get() ) != EOF ) {

      switch ( grade ) {      // switch nested in while

         case 'A':  // grade was uppercase A
         case 'a':  // or lowercase a
            ++aCount;
            break;  // necessary to exit switch

         case 'B':  // grade was uppercase B
         case 'b':  // or lowercase b
            ++bCount;
            break;

         case 'C':  // grade was uppercase C
         case 'c':  // or lowercase c
            ++cCount;
            break;

         case 'D':  // grade was uppercase D
         case 'd':  // or lowercase d
            ++dCount;
            break;

         case 'F':  // grade was uppercase F
         case 'f':  // or lowercase f
            ++fCount;
            break;
```

```

         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ':  // and spaces in input
            break;

```

Fig. 2.22 An example using **switch** (part 1 of 2).

```
         default:   // catch all other characters
            cout << "Incorrect letter grade entered."
                 << " Enter a new grade." << endl;
            break;  // optional
      }
   }

   cout << "\n\nTotals for each letter grade are:"
        << "\nA: " << aCount
        << "\nB: " << bCount
        << "\nC: " << cCount
        << "\nD: " << dCount
        << "\nF: " << fCount << endl;

   return 0;
}
```

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 2.22 An example using **switch** (part 2 of 2).

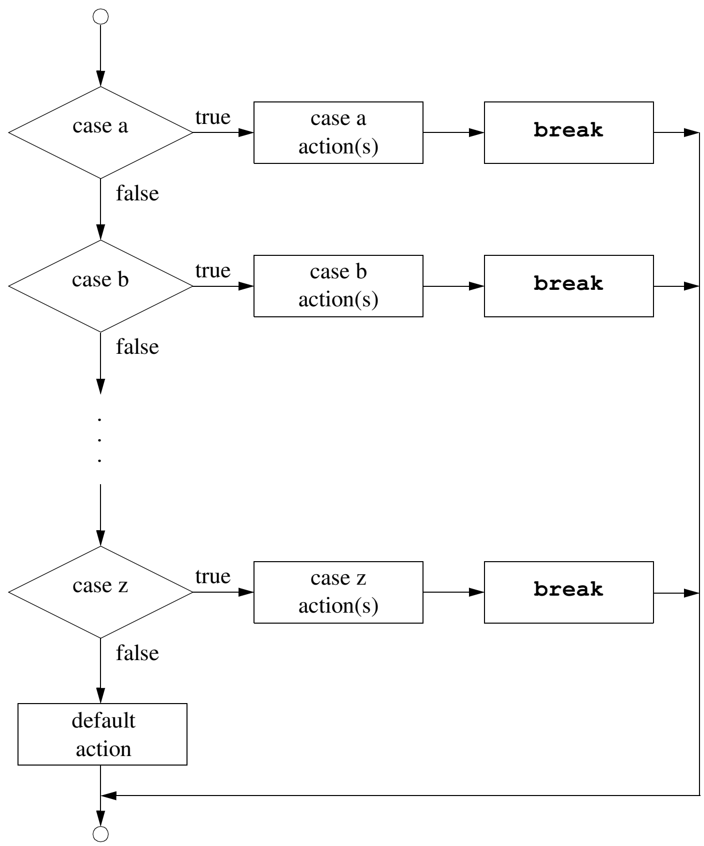


Fig. 2.23 The **switch** multiple-selection structure with **break**s.

```
// Fig. 2.24: fig02_24.cpp
// Using the do/while repetition structure
#include <iostream.h>

int main()
{
   int counter = 1;

   do {
      cout << counter << "  ";
   } while ( ++counter <= 10 );

   cout << endl;

   return 0;
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 2.24 Using the **do/while** structure.

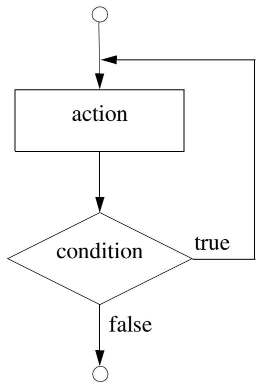


Fig. 2.25 Flowcharting the **do/while** repetition structure.

```
// Fig. 2.26: fig02_26.cpp
// Using the break statement in a for structure
#include <iostream.h>

int main()
{
   // x declared here so it can be used after the loop
   int x;

   for ( x = 1; x <= 10; x++ ) {

      if ( x == 5 )
         break;    // break loop only if x is 5

      cout << x << " ";
   }

   cout << "\nBroke out of loop at x of " << x << endl;
   return 0;
}
```

Fig. 2.26 Using the **break** statement in a **for** structure (part 1 of 2).

```
1 2 3 4
Broke out of loop at x of 5
```

Fig. 2.26 Using the **break** statement in a **for** structure (part 2 of 2).

```
// Fig. 2.27: fig02_07.cpp
// Using the continue statement in a for structure
#include <iostream.h>

int main()
{
   for ( int x = 1; x <= 10; x++ ) {

      if ( x == 5 )
         continue;  // skip remaining code in loop
                    // only if x is 5

      cout << x << " ";
   }

   cout << "\nUsed continue to skip printing the value 5"
        << endl;
   return 0;
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 2.27 Using the `continue` statement in a `for` structure.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 2.28 Truth table for the `&&` (logical AND) operator.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 2.29 Truth table for the `||` (logical OR) operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. 2.30 Truth table for operator `!` (logical negation).

Operators	Associativity	Type
`()`	left to right	parentheses
`++ -- + - ! static_cast<type>()`	right to left	unary
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`<< >>`	left to right	insertion/extraction
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment
`,`	left to right	comma

Fig. 2.31 Operator precedence and associativity.

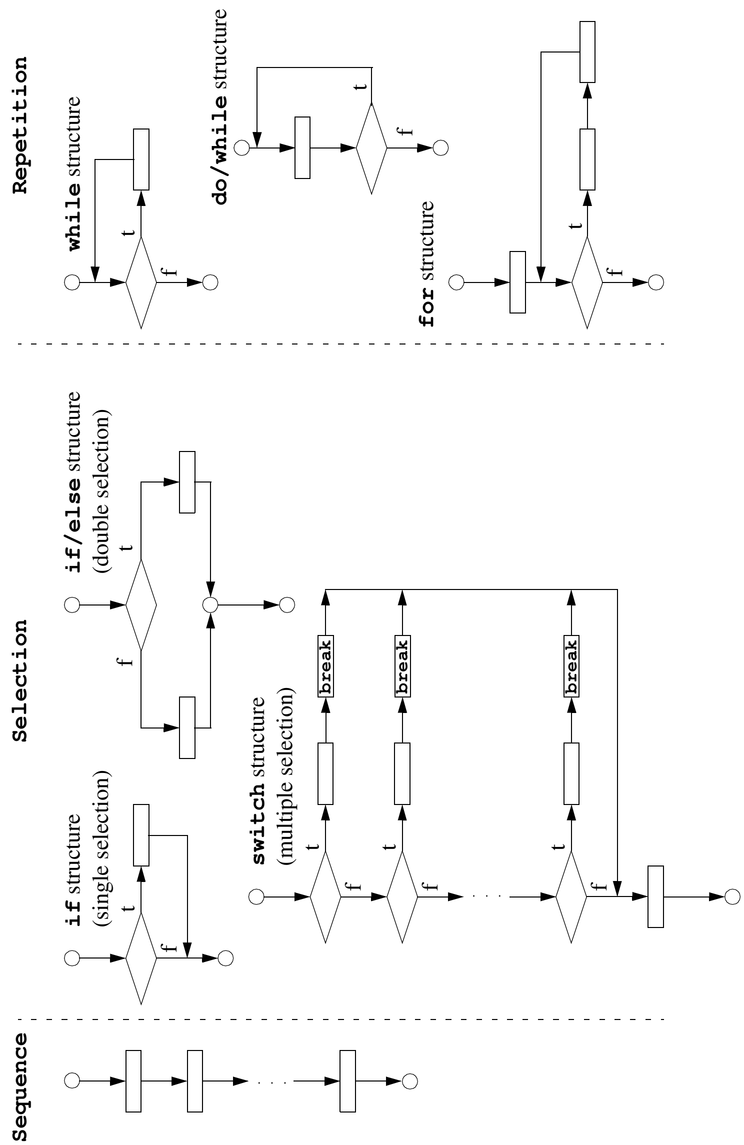


Fig. 2.32 C++'s single-entry/single-exit sequence, selection, and repetition structures.

Rules for Forming Structured Programs

1) Begin with the "simplest flowchart" (Fig. 2.34).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3) Any rectangle (action) can be replaced by any control structure (sequence, `if`, `if/else`, `switch`, `while`, `do/while`, or `for`).
4) Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 2.33 Rules for forming structured programs.

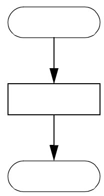


Fig. 2.34 The simplest flowchart.

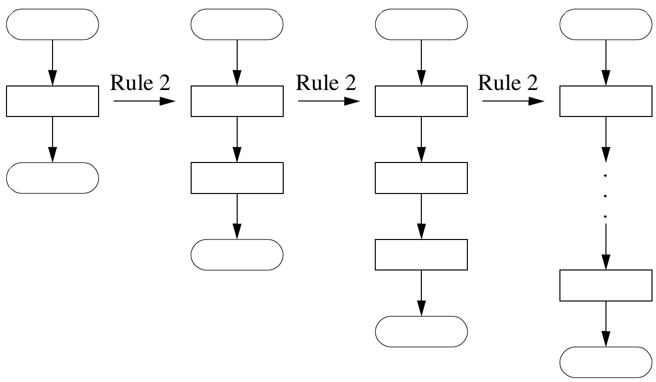


Fig. 2.35 Repeatedly applying rule 2 of Fig. 2.33 to the simplest flowchart.

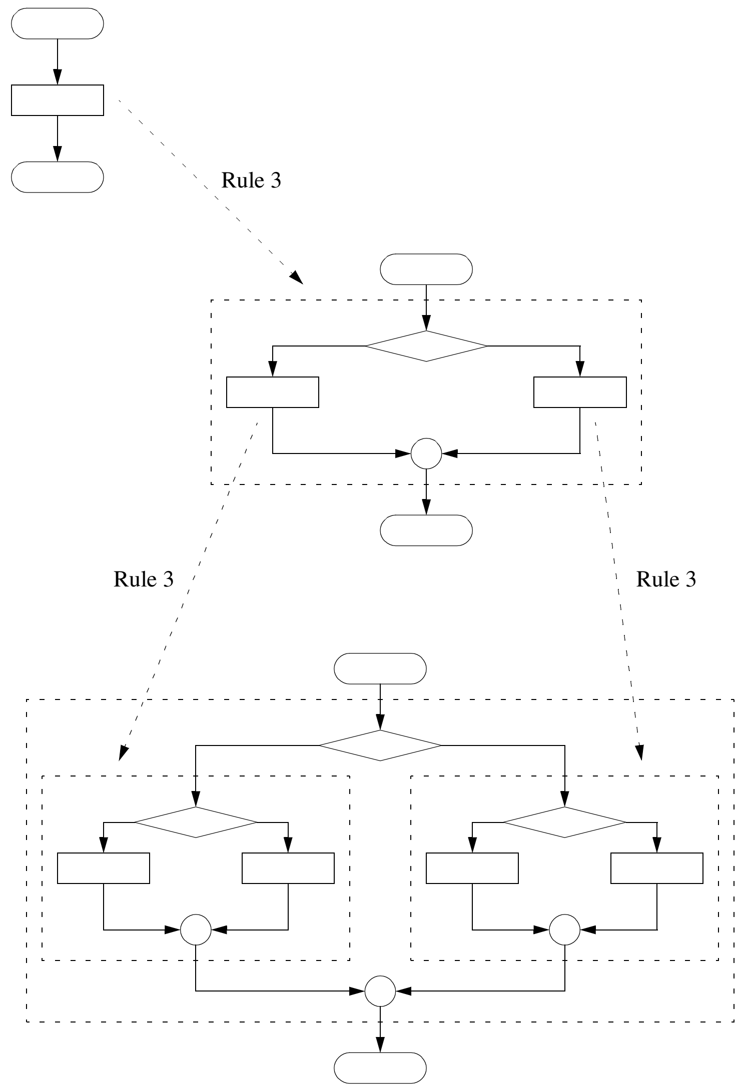


Fig. 2.36 Applying rule 3 of Fig. 2.33 to the simplest flowchart.

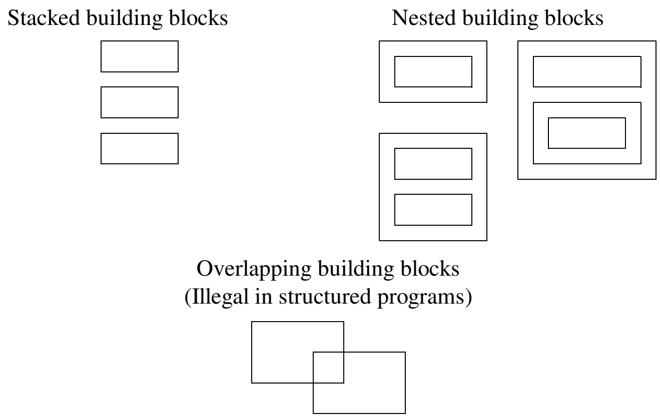


Fig. 2.37 Stacked, nested, and overlapped building blocks.

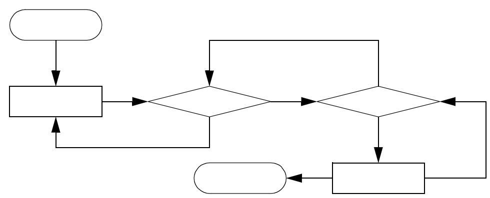


Fig. 2.38 An unstructured flowchart.

Illustrations List *(Main Page)*

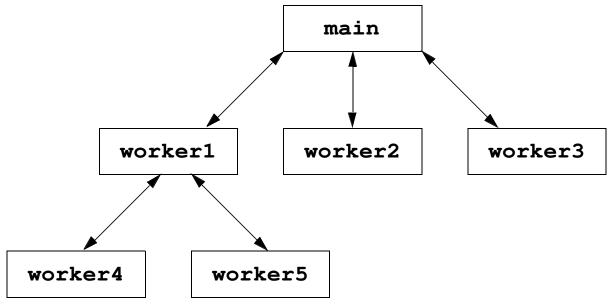


Fig. 3.1 Hierarchical boss function/worker function relationship.

Function	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`fabs( x )`	absolute value of *x*	if `x > 0` then `abs( x )` is `x` if `x = 0` then `abs( x )` is `0.0` if `x < 0` then `abs( x )` is `x`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`fmod( x, y )`	remainder of *x/y* as a floating point number	`fmod( 13.657, 2.333 )` is `1.992`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`log10( x )`	logarithm of *x* (base 10)	`log( 10.0 )` is `1.0` `log( 100.0 )` is `2.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128` `pow( 9, .5 )` is `3`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Fig. 3.2 Commonly used math library functions.

```
// Fig. 3.3: fig03_03.cpp
// Creating and using a programmer-defined function
#include <iostream.h>

int square( int );   // function prototype

int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";

   cout << endl;
   return 0;
}

// Function definition
int square( int y )
{
   return y * y;
}
```

```
 1  4  9  16  25  36  49  64  81  100
```

Fig. 3.3 Creating and using a programmer-defined function.

```
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream.h>

int maximum( int, int, int );   // function prototype

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;
```

Fig. 3.4 Programmer-defined **maximum** function (part 1 of 2).

```

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;

   return 0;
}

// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

```

```
    return max;
}
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 92 35 14
Maximum is: 92
```

```
Enter three integers: 45 19 98
Maximum is: 98
```

Fig. 3.4 Programmer-defined **`maximum`** function (part 2 of 2).

Data types	
`long double`	
`double`	
`float`	
`unsigned long int`	(synonymous with **`unsigned long`**)
`long int`	(synonymous with **`long`**)
`unsigned int`	(synonymous with **`unsigned`**)
`int`	
`unsigned short int`	(synonymous with **`unsigned short`**)
`short int`	(synonymous with **`short`**)
`unsigned char`	
`short`	
`char`	

Fig. 3.5 Promotion hierarchy for built-in data types.

Standard library header file	Explanation
Old-style header files (used early in the book)	
`<assert.h>`	Contains macros and information for adding diagnostics that aid program debugging. The new version of this header file is **`<cassert>`**.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. The new version of this header file is **`<cctype>`**.

Fig. 3.6 Standard library header files (part 1 of 3).

Standard library header file	Explanation
<float.h>	Contains the floating-point size limits of the system. The new version of this header file is **<cfloat>**.
<limits.h>	Contains the integral size limits of the system. The new version of this header file is **<climits>**.
<math.h>	Contains function prototypes for math library functions. The new version of this header file is **<cmath>**.
<stdio.h>	Contains function prototypes for the standard input/output library functions and information used by them. The new version of this header file is **<cstdio>**.
<stdlib.h>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions. The new version of this header file is **<cstdlib>**.
<string.h>	Contains function prototypes for C-style string processing functions. The new version of this header file is **<cstring>**.
<time.h>	Contains function prototypes and types for manipulating the time and date. The new version of this header file is **<ctime>**.
<iostream.h>	Contains function prototypes for the standard input and standard output functions. The new version of this header file is **<iostream>**.
<iomanip.h>	Contains function prototypes for the stream manipulators that enable formatting of streams of data. The new version of this header file is **<iomanip>**.
<fstream.h>	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 14). The new version of this header file is **<fstream>**.
New-style header files (used later in the book)	
<utility>	Contains classes and functions that are used by many standard library header files.
<vector>, **<list>**, **<deque>**, **<queue>**, **<stack>**, **<map>**, **<set>**, **<bitset>**	The header files contain classes that implement the standard library containers. Containers are use to store data during a program's execution. We discuss these header files in the chapter entitled "The Standard Template Library."
<functional>	Contains classes and functions used by algorithms of the standard library.
<memory>	Contains classes and functions used by the standard library to allocate memory to the standard library containers.
<iterator>	Contains classes for manipulating data in the standard library containers.
<algorithm>	Contains functions for manipulating data in the standard library containers.
<exception> **<stdexcept>**	These header files contain classes that are used for exception handling (discussed in Chapter 13).
<string>	Contains the definition of class **string** from the standard library (discussed in Chapter 19, "Strings").
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 14).

Fig. 3.6 Standard library header files (part 2 of 3).

Standard library header file	Explanation
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<limits>	Contains a class for defining the numerical data type limits on each computer platform.
<typeinfo>	Contains classes for run-time type identification (determining data types at execution time).

Fig. 3.6 Standard library header files (part 3 of 3).

```
// Fig. 3.7: fig03_07.cpp
// Shifted, scaled integers produced by 1 + rand() % 6
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
   for ( int i = 1; i <= 20; i++ ) {
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

```
         5         5         3         5         5
         2         4         2         5         5
         5         3         2         2         1
         5         1         4         6         4
```

Fig. 3.7 Shifted, scaled integers produced by **1 + rand() % 6**.

```
// Fig. 3.8: fig03_08.cpp
// Roll a six-sided die 6000 times
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
   int frequency1 = 0, frequency2 = 0,
       frequency3 = 0, frequency4 = 0,
       frequency5 = 0, frequency6 = 0,
       face;

   for ( int roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6;

      switch ( face ) {
         case 1:
            ++frequency1;
            break;
         case 2:
            ++frequency2;
            break;
         case 3:
            ++frequency3;
            break;
         case 4:
            ++frequency4;
            break;
         case 5:
            ++frequency5;
            break;
         case 6:
            ++frequency6;
            break;
         default:
            cout << "should never get here!";
      }
   }

   cout << "Face" << setw( 13 ) << "Frequency"
        << "\n   1" << setw( 13 ) << frequency1
        << "\n   2" << setw( 13 ) << frequency2
        << "\n   3" << setw( 13 ) << frequency3
        << "\n   4" << setw( 13 ) << frequency4
        << "\n   5" << setw( 13 ) << frequency5
        << "\n   6" << setw( 13 ) << frequency6 << endl;

   return 0;
}
```

Fig. 3.8 Rolling a six-sided die 6000 times (part 1 of 2).

```
Face    Frequency
1             987
2             984
3            1029
4             974
5            1004
6           1022
```

Fig. 3.8 Rolling a six-sided die 6000 times (part 2 of 2).

```
// Fig. 3.9: fig03_09.cpp
// Randomizing die-rolling program
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
   unsigned seed;

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed );

   for ( int i = 1; i <= 10; i++ ) {
      cout << setw( 10 ) << 1 + rand() % 6;

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

```
Enter seed: 67
1         6         5         1         4
5         6         3         1         2
```

```
Enter seed: 432
4         2         6         4         3
2         5         1         4         4
```

```
Enter seed: 67
1         6         5         1         4
5         6         3         1         2
```

Fig. 3.9 Randomizing the die-rolling program.

```
// Fig. 3.10: fig03_10.cpp
// Craps
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

int rollDice( void );   // function prototype

int main()
{
   enum Status { CONTINUE, WON, LOST };
   int sum, myPoint;
   Status gameStatus;

   srand( time( NULL ) );
   sum = rollDice();           // first roll of the dice

   switch ( sum ) {
      case 7:
      case 11:                 // win on first roll
         gameStatus = WON;
         break;
      case 2:
```

```
      case 3:
      case 12:                  // lose on first roll
         gameStatus = LOST;
         break;
      default:                  // remember point
         gameStatus = CONTINUE;
         myPoint = sum;
         cout << "Point is " << myPoint << endl;
         break;                 // optional
   }

   while ( gameStatus == CONTINUE ) {    // keep rolling
      sum = rollDice();

      if ( sum == myPoint )       // win by making point
         gameStatus = WON;
      else
         if ( sum == 7 )          // lose by rolling 7
            gameStatus = LOST;
   }

   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;

   return 0;
}
```

Fig. 3.10 Program to simulate the game of craps (part 1 of 2).

```
int rollDice( void )
{
   int die1, die2, workSum;

   die1 = 1 + rand() % 6;
   die2 = 1 + rand() % 6;
   workSum = die1 + die2;
   cout << "Player rolled " << die1 << " + " << die2
        << " = " << workSum << endl;

   return workSum;
}
```

Fig. 3.10 Program to simulate the game of craps (part 2 of 2).

```
Player rolled 6 + 5 = 11
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Fig. 3.11 Sample runs for the game of craps.

```
// Fig. 3.12: fig03_12.cpp
// A scoping example
#include <iostream.h>

void a( void );    // function prototype
void b( void );    // function prototype
void c( void );    // function prototype
```

Fig. 3.12 A scoping example (part 1 of 3).

```

int x = 1;       // global variable

int main()
{
   int x = 5;    // local variable to main

   cout << "local x in outer scope of main is " << x << endl;

   {             // start new scope
      int x = 7;

      cout << "local x in inner scope of main is " << x << endl;
   }             // end new scope

   cout << "local x in outer scope of main is " << x << endl;

   a();          // a has automatic local x
   b();          // b has static local x
   c();          // c uses global x
   a();          // a reinitializes automatic local x
```

```
   b();          // static local x retains its previous value
   c();          // global x also retains its value

   cout << "local x in main is " << x << endl;

   return 0;
}

void a( void )
{
   int x = 25;  // initialized each time a is called

   cout << endl << "local x in a is " << x
        << " after entering a" << endl;
   ++x;
   cout << "local x in a is " << x
        << " before exiting a" << endl;
}

void b( void )
{
    static int x = 50;  // Static initialization only
                        // first time b is called.
    cout << endl << "local static x is " << x
         << " on entering b" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting b" << endl;
}

```

Fig. 3.12 A scoping example (part 2 of 3).

```
void c( void )
{
   cout << endl << "global x is " << x
        << " on entering c" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting c" << endl;
}
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Fig. 3.12 A scoping example (part 3 of 3).

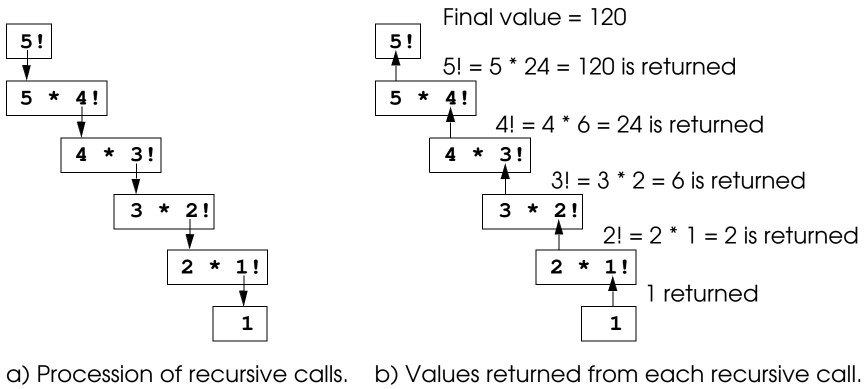


Fig. 3.13 Recursive evaluation of 5!.

```
// Fig. 3.14: fig03_14.cpp
// Recursive factorial function
#include <iostream.h>
#include <iomanip.h>

unsigned long factorial( unsigned long );

int main()
{
   for ( int i = 0; i <= 10; i++ )
      cout << setw( 2 ) << i << "! = " << factorial( i ) << endl;

   return 0;
}

// Recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
   if ( number <= 1 )   // base case
      return 1;
   else                 // recursive case
      return number * factorial( number - 1 );
}
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 3.14 Calculating factorials with a recursive function.

```
// Fig. 3.15: fig03_15.cpp
// Recursive fibonacci function
#include <iostream.h>

long fibonacci( long );

int main()
{
    long result, number;

    cout << "Enter an integer: ";
    cin >> number;
    result = fibonacci( number );
    cout << "Fibonacci(" << number << ") = " << result << endl;
    return 0;
}

// Recursive definition of function fibonacci
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )  // base case
      return n;
   else                     // recursive case
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

Fig. 3.15 Recursively generating Fibonacci numbers (part 1 of 2).

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Fig. 3.15 Recursively generating Fibonacci numbers (part 2 of 2).

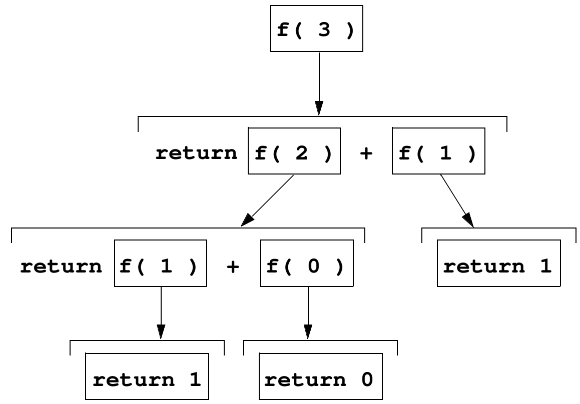


Fig. 3.16 Set of recursive calls to method **fibonacci**.

Chapter	Recursion Examples and Exercises
Chapter 3	Factorial function
	Fibonacci function
	Greatest common divisor
	Sum of two integers
	Multiply two integers
	Raising an integer to an integer power
	Towers of Hanoi
	Printing keyboard inputs in reverse
	Visualizing recursion
Chapter 4	Sum the elements of an array
	Print an array
	Print an array backwards
	Print a string backwards
	Check if a string is a palindrome
	Minimum value in an array
	Selection sort
	Eight Queens
	Linear search
	Binary search
Chapter 5	Quicksort
	Maze traversal
	Printing a string input at the keyboard backwards
Chapter 15	Linked list insert
	Linked list delete
	Search a linked list
	Print a linked list backwards

Fig. 3.17 Summary of recursion examples and exercises in the text (part 1 of 2).

Chapter	Recursion Examples and Exercises
	Binary tree insert
	Preorder traversal of a binary tree
	Inorder traversal of a binary tree
	Postorder traversal of a binary tree

Fig. 3.17 Summary of recursion examples and exercises in the text (part 2 of 2).

```
// Fig. 3.18: fig03_18.cpp
// Functions that take no arguments
#include <iostream.h>

void function1();
void function2( void );

int main()
{
   function1();
   function2();

   return 0;
}

void function1()
{
   cout << "function1 takes no arguments" << endl;
}

void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
}
```

```
function1 takes no arguments
function2 also takes no arguments
```

Fig. 3.18 Two ways to declare and use functions that take no arguments.

```
// Fig. 3.19: fig03_19.cpp
// Using an inline function to calculate
// the volume of a cube.
#include <iostream.h>

inline float cube( const float s ) { return s * s * s; }

int main()
{
   cout << "Enter the side length of your cube:  ";

   float side;

   cin >> side;
   cout << "Volume of cube with side "
        << side << " is " << cube( side ) << endl;

   return 0;
}
```

```
Enter the side length of your cube:  3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 3.19 Using an **inline** function to calculate the volume of a cube.

```
// Fig. 3.20: fig03_20.cpp
// Comparing call-by-value and call-by-reference
// with references.
#include <iostream.h>

int squareByValue( int );
void squareByReference( int & );

int main()
{
   int x = 2, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue( x ) << endl
        << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;

   return 0;
}
```

Fig. 3.20 An example of call-by-reference (part 1 of 2).

```

int squareByValue( int a )
{
   return a *= a;    // caller's argument not modified
}

void squareByReference( int &cRef )
{
   cRef *= cRef;     // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 3.20 An example of call-by-reference (part 2 of 2).

```
// References must be initialized
#include <iostream.h>

int main()
{
   int x = 3, &y = x;  // y is now an alias for x

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;
}
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 3.21 Using an initialized reference.

```
// References must be initialized
#include <iostream.h>

int main()
{
   int x = 3, &y;    // Error: y must be initialized

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;
}
```

```
Compiling FIG03_21.CPP:
Error FIG03_21.CPP 6: Reference variable 'y' must be
   initialized
```

Fig. 3.22 Attempting to use an uninitialized reference.

```
// Fig. 3.23: fig03_23.cpp
// Using default arguments
#include <iostream.h>

int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
   cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
        << endl;

   return 0;
}

// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 3.23 Using default arguments.

```
// Fig. 3.24: fig03_24.cpp
// Using the unary scope resolution operator
#include <iostream.h>
#include <iomanip.h>

const double PI = 3.14159265358979;

int main()
{
   const float PI = static_cast< float >( ::PI );

   cout << setprecision( 20 )
        << "  Local float value of PI = " << PI
        << "\nGlobal double value of PI = " << ::PI << endl;

   return 0;
}
```

```
  Local float value of PI = 3.14159
Global double value of PI = 3.14159265358979
```

Fig. 3.24 Using the unary scope resolution operator.

```
// Fig. 3.25: fig03_25.cpp
// Using overloaded functions
#include <iostream.h>

int square( int x ) { return x * x; }

double square( double y ) { return y * y; }

int main()
{
   cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

   return 0;
}
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Fig. 3.25 Using overloaded functions.

```
// Name mangling
int square(int x) { return x * x; }

double square(double y) { return y * y; }

void nothing1(int a, float b, char c, int *d)
   { }  // empty function body

char *nothing2(char a, int b, float *c, double *d)
   { return 0; }

int main()
{
   return 0;
}
```

```
public   _main
public   @nothing2$qzcipfpd
public   @nothing1$qifzcpi
public   @square$qd
public   @square$qi
```

Fig. 3.26 Name mangling to enable type-safe linkage.

```
// Fig. 3.27: fig03_27.cpp
// Using a function template
#include <iostream.h>

template < class T >
T maximum( T value1, T value2, T value3 )
{
   T max = value1;

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   return max;
}

int main()
{
   int int1, int2, int3;

   cout << "Input three integer values: ";
   cin >> int1 >> int2 >> int3;
   cout << "The maximum integer value is: "
        << maximum( int1, int2, int3 );        // int version
```

Fig. 3.27 Using a function template (part 1 of 2).

```

   double double1, double2, double3;

   cout << "\nInput three double values: ";
   cin >> double1 >> double2 >> double3;
   cout << "The maximum double value is: "
      << maximum( double1, double2, double3 ); // double version

   char char1, char2, char3;

   cout << "\nInput three characters: ";
   cin >> char1 >> char2 >> char3;
   cout << "The maximum character value is: "
        << maximum( char1, char2, char3 )      // char version
        << endl;

   return 0;
}
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
Input three characters: A C B
The maximum character value is: C
```

Fig. 3.27 Using a function template (part 2 of 2).

Illustrations List *(Main Page)*

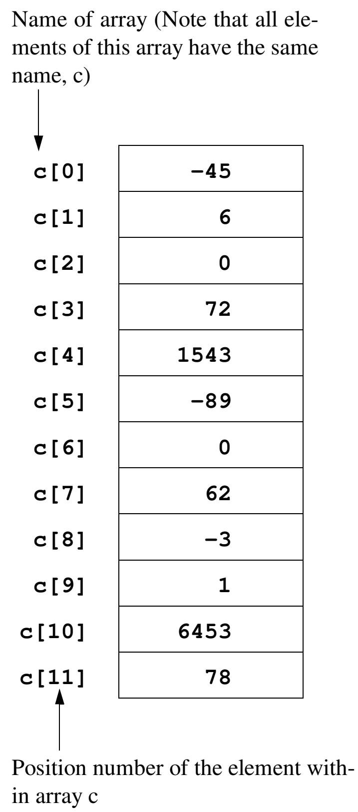


Fig. 4.1 A 12-element array.

Operators	Associativity	Type
`()` `[]`	left to right	highest
`++` `--` `+` `-` `!` `static_cast<`*type*`>()`	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 4.2 Operator precedence and associativity.

```
// Fig. 4.3: fig04_03.cpp
// initializing an array
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int i, n[ 10 ];

   for ( i = 0; i < 10; i++ )       // initialize array
      n[ i ] = 0;

   cout << "Element" << setw( 13 ) << "Value" << endl;

   for ( i = 0; i < 10; i++ )       // print array
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;
}
```

Fig. 4.3 Initializing the elements of an array to zeros (part 1 of 2).

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

Fig. 4.3 Initializing the elements of an array to zeros (part 2 of 2).

```
// Fig. 4.4: fig04_04.cpp
// Initializing an array with a declaration
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;
}
```

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

Fig. 4.4 Initializing the elements of an array with a declaration.

```
// Fig. 4.5: fig04_05.cpp
// Initialize array s to the even integers from 2 to 20.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   const int arraySize = 10;
   int j, s[ arraySize ];

   for ( j = 0; j < arraySize; j++ )   // set the values
      s[ j ] = 2 + 2 * j;
```

Fig. 4.5 Generating values to be placed into elements of an array (part 1 of 2).

```

   cout << "Element" << setw( 13 ) << "Value" << endl;

   for ( j = 0; j < arraySize; j++ )   // print the values
      cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;

   return 0;
}
```

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

Fig. 4.5 Generating values to be placed into elements of an array (part 2 of 2).

```
// Fig. 4.6: fig04_06.cpp
// Using a properly initialized constant variable
#include <iostream.h>

int main()
{
   const int x = 7;  // initialized constant variable

   cout << "The value of constant variable x is: "
        << x << endl;

   return 0;
}
```

Fig. 4.6 Correctly initializing and using a constant variable (part 1 of 2).

```
The value of constant variable x is: 7
```

Fig. 4.6 Correctly initializing and using a constant variable (part 2 of 2).

```
// Fig. 4.7: fig04_07.cpp
// A const object must be initialized

int main()
{
   const int x;  // Error: x must be initialized

   x = 7;        // Error: cannot modify a const variable

   return 0;
}
```

```
Compiling FIG04_7.CPP:
Error FIG04_7.CPP 6: Constant variable 'x' must be
   initialized
Error FIG04_7.CPP 8: Cannot modify a const object
```

Fig. 4.7 A `const` object must be initialized.

```
// Fig. 4.8: fig04_08.cpp
// Compute the sum of the elements of the array
#include <iostream.h>

int main()
{
   const int arraySize = 12;
   int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99,
                          16, 45, 67, 89, 45 };
   int total = 0;

   for ( int i = 0; i < arraySize ; i++ )
      total += a[ i ];

   cout << "Total of array element values is " << total << endl;
   return 0;
}
```

Fig. 4.8 Computing the sum of the elements of an array (part 1 of 2).

```
Total of array element values is 383
```

Fig. 4.8 Computing the sum of the elements of an array (part 2 of 2).

```
// Fig. 4.9: fig04_09.cpp
// Student poll program
#include <iostream.h>
#include <iomanip.h>

int main()
{
   const int responseSize = 40, frequencySize = 11;
   int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
   int frequency[ frequencySize ] = { 0 };

   for ( int answer = 0; answer < responseSize; answer++ )
      ++frequency[ responses[ answer ] ];

   cout << "Rating" << setw( 17 ) << "Frequency" << endl;

   for ( int rating = 1; rating < frequencySize; rating++ )
      cout << setw( 6 ) << rating
           << setw( 17 ) << frequency[ rating ] << endl;

   return 0;
}
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

Fig. 4.9 A student poll analysis program.

```
// Fig. 4.10: fig04_10.cpp
// Histogram printing program
#include <iostream.h>
#include <iomanip.h>

int main()
{
   const int arraySize = 10;
   int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };

   cout << "Element" << setw( 13 ) << "Value"
        << setw( 17 ) << "Histogram" << endl;

   for ( int i = 0; i < arraySize ; i++ ) {
      cout << setw( 7 ) << i << setw( 13 )
           << n[ i ] << setw( 9 );

      for ( int j = 0; j < n[ i ]; j++ )   // print one bar
         cout << '*';

      cout << endl;
   }

   return 0;
}
```

Fig. 4.10 A program that prints histograms (part 1 of 2).

```
Element        Value          Histogram
      0           19          *******************
      1            3          ***
      2           15          ***************
      3            7          *******
      4           11          ***********
      5            9          *********
      6           13          *************
      7            5          *****
      8           17          *****************
      9            1          *
```

Fig. 4.10 A program that prints histograms (part 2 of 2).

```
// Fig. 4.11: fig04_11.cpp
// Roll a six-sided die 6000 times
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   const int arraySize = 7;
   int face, frequency[ arraySize ] = { 0 };

   srand( time( 0 ) );

   for ( int roll = 1; roll <= 6000; roll++ )
      ++frequency[ 1 + rand() % 6 ]; // replaces 20-line switch
                                     // of Fig. 3.8

   cout << "Face" << setw( 13 ) << "Frequency" << endl;

```

Fig. 4.11 Dice-rolling program using arrays instead of `switch` (part 1 of 2).

```
   // ignore element 0 in the frequency array
   for ( face = 1; face < arraySize ; face++ )
      cout << setw( 4 ) << face
           << setw( 13 ) << frequency[ face ] << endl;

   return 0;
}
```

```
Face    Frequency
   1         1037
   2          987
   3         1013
   4         1028
   5          952
   6          983
```

Fig. 4.11 Dice-rolling program using arrays instead of `switch` (part 2 of 2)

```
// Fig. 4_12: fig04_12.cpp
// Treating character arrays as strings
#include <iostream.h>

int main()
{
   char string1[ 20 ], string2[] = "string literal";

   cout << "Enter a string: ";
   cin >> string1;
   cout << "string1 is: " << string1
        << "\nstring2 is: " << string2
        << "string1 with spaces between characters is:\n";

   for ( int i = 0; string1[ i ] != '\0'; i++ )
      cout << string1[ i ] << ' ';

   cin >> string1;  // reads "there"
   cout << "\nstring1 is: " << string1 << endl;

   cout << endl;
   return 0;
}
```

Fig. 4.12 Treating character arrays as strings (part 1 of 2).

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
string1 is: there
```

Fig. 4.12 Treating character arrays as strings (part 2 of 2).

```
// Fig. 4.13: fig04_13.cpp
// Static arrays are initialized to zero
#include <iostream.h>

void staticArrayInit( void );
void automaticArrayInit( void );
```

Fig. 4.13 Comparing **`static`** array initialization and automatic array initialization (part 1 of 3).

```

int main()
{
   cout << "First call to each function:\n";
   staticArrayInit();
   automaticArrayInit();

   cout << "\n\nSecond call to each function:\n";
   staticArrayInit();
   automaticArrayInit();
   cout << endl;

   return 0;
}

// function to demonstrate a static local array
void staticArrayInit( void )
{
   static int array1[ 3 ];
   int i;

   cout << "\nValues on entering staticArrayInit:\n";

   for ( i = 0; i < 3; i++ )
      cout << "array1[" << i << "] = " << array1[ i ] << "  ";

   cout << "\nValues on exiting staticArrayInit:\n";

   for ( i = 0; i < 3; i++ )
      cout << "array1[" << i << "] = "
           << ( array1[ i ] += 5 ) << "  ";
}

// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
   int i, array2[ 3 ] = { 1, 2, 3 };

   cout << "\n\nValues on entering automaticArrayInit:\n";

   for ( i = 0; i < 3; i++ )
      cout << "array2[" << i << "] = " << array2[ i ] << "  ";

   cout << "\nValues on exiting automaticArrayInit:\n";

   for ( i = 0; i < 3; i++ )
      cout << "array2[" << i << "] = "
           << ( array2[ i ] += 5 ) << "  ";
}
```

Fig. 4.13 Comparing **`static`** array initialization and automatic array initialization (part 2 of 3).

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Fig. 4.13 Comparing `static` array initialization and automatic array initialization (part 3 of 3).

```
// Fig. 4.14: fig04_14.cpp
// Passing arrays and individual array elements to functions
#include <iostream.h>
#include <iomanip.h>

void modifyArray( int [], int );  // appears strange
void modifyElement( int );

int main()
{
   const int arraySize = 5;
   int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };

   cout << "Effects of passing entire array call-by-reference:"
        << "\n\nThe values of the original array are:\n";

   for ( i = 0; i < arraySize; i++ )
      cout << setw( 3 ) << a[ i ];

   cout << endl;

   // array a passed call-by-reference
   modifyArray( a, arraySize );

   cout << "The values of the modified array are:\n";

   for ( i = 0; i < arraySize; i++ )
      cout << setw( 3 ) << a[ i ];

   cout << "\n\n\n"
        << "Effects of passing array element call-by-value:"
```

```
         << "\n\nThe value of a[3] is " << a[ 3 ] << '\n';
```

Fig. 4.14 Passing arrays and individual array elements to functions (part 1 of 2).

```

   modifyElement( a[ 3 ] );

   cout << "The value of a[3] is " << a[ 3 ] << endl;

   return 0;
}

void modifyArray( int b[], int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; j++ )
      b[ j ] *= 2;
}

void modifyElement( int e )
{
   cout << "Value in modifyElement is "
        << ( e *= 2 ) << endl;
}
```

```
Effects of passing entire array call-by-reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element call-by-value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Fig. 4.14 Passing arrays and individual array elements to functions (part 2 of 2).

```
// Fig. 4.15: fig04_15.cpp
// Demonstrating the const type qualifier
#include <iostream.h>

void tryToModifyArray( const int [] );

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );
   cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
   return 0;
}

void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2;    // error
   b[ 1 ] /= 2;    // error
   b[ 2 ] /= 2;    // error
}
```

```
Compiling FIG04_15.CPP:
Error FIG04_15.CPP 18: Cannot modify a const object
Error FIG04_15.CPP 19: Cannot modify a const object
Error FIG04_15.CPP 20: Cannot modify a const object
Warning FIG04_15.CPP 21: Parameter 'b' is never used
```

Fig. 4.15 Demonstrating the `const` type qualifier.

```
// Fig. 4.16: fig04_16.cpp
// This program sorts an array's values into
// ascending order
#include <iostream.h>
#include <iomanip.h>

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int i, hold;

   cout << "Data items in original order\n";

   for ( i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << a[ i ];

   for ( int pass = 0; pass < arraySize - 1; pass++ ) // passes

      for ( i = 0; i < arraySize - 1; i++ )      // one pass

         if ( a[ i ] > a[ i + 1 ] ) {      // one comparison
            hold = a[ i ];                 // one swap
            a[ i ] = a[ i + 1 ];
            a[ i + 1 ] = hold;
         }

   cout << "\nData items in ascending order\n";

```

Fig. 4.16 Sorting an array with bubble sort (part 1 of 2).

```
   for ( i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << a[ i ];

   cout << endl;
   return 0;
}
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 4.16 Sorting an array with bubble sort (part 2 of 2).

```
// Fig. 4.17: fig04_17.cpp
// This program introduces the topic of survey data analysis.
// It computes the mean, median, and mode of the data.
#include <iostream.h>
#include <iomanip.h>

void mean( const int [], int );
void median( int [], int );
void mode( int [], int [], int );
void bubbleSort( int[], int );
void printArray( const int[], int );

int main()
{
   const int responseSize = 99;
   int frequency[ 10 ] = { 0 },
       response[ responseSize ] =
          { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
            7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
            6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
            7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
            6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
            7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
            5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
            7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
            7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
            4, 5, 6, 1, 6, 5, 7, 8, 7 };

   mean( response, responseSize );
   median( response, responseSize );
   mode( frequency, response, responseSize );

   return 0;
}

void mean( const int answer[], int arraySize )
{
   int total = 0;

   cout << "********\n  Mean\n********\n";

   for ( int j = 0; j < arraySize; j++ )
      total += answer[ j ];

```

Fig. 4.17 Survey data analysis program (part 1 of 3).

```
   cout << "The mean is the average value of the data\n"
        << "items. The mean is equal to the total of\n"
        << "all the data items divided by the number\n"
        << "of data items (" << arraySize
        << "). The mean value for\nthis run is: "
        << total << " / " << arraySize << " = "
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 4 ) << ( float ) total / arraySize
        << "\n\n";
}

void median( int answer[], int size )
{
   cout << "\n********\n Median\n********\n"
```

```
        << "The unsorted array of responses is";

   printArray( answer, size );
   bubbleSort( answer, size );
   cout << "\n\nThe sorted array is";
   printArray( answer, size );
   cout << "\n\nThe median is element " << size / 2
        << " of\nthe sorted " << size
        << " element array.\nFor this run the median is "
        << answer[ size / 2 ] << "\n\n";
}

void mode( int freq[], int answer[], int size )
{
   int rating, largest = 0, modeValue = 0;

   cout << "\n********\n  Mode\n********\n";

   for ( rating = 1; rating <= 9; rating++ )
      freq[ rating ] = 0;

   for ( int j = 0; j < size; j++ )
      ++freq[ answer[ j ] ];

   cout << "Response"<< setw( 11 ) << "Frequency"
        << setw( 19 ) << "Histogram\n\n" << setw( 55 )
        << "1    1    2    2\n" << setw( 56 )
        << "5    0    5    0    5\n\n";

   for ( rating = 1; rating <= 9; rating++ ) {
      cout << setw( 8 ) << rating << setw( 11 )
           << freq[ rating ] << "          ";

      if ( freq[ rating ] > largest ) {
         largest = freq[ rating ];
         modeValue = rating;
      }
```

Fig. 4.17 Survey data analysis program (part 2 of 3).

```

      for ( int h = 1; h <= freq[ rating ]; h++ )
         cout << '*';

      cout << '\n';
   }

   cout << "The mode is the most frequent value.\n"
        << "For this run the mode is " << modeValue
        << " which occurred " << largest << " times." << endl;
}

void bubbleSort( int a[], int size )
{
   int hold;

   for ( int pass = 1; pass < size; pass++ )

      for ( int j = 0; j < size - 1; j++ )

         if ( a[ j ] > a[ j + 1 ] ) {
            hold = a[ j ];
            a[ j ] = a[ j + 1 ];
```

```
            a[ j + 1 ] = hold;
         }
}

void printArray( const int a[], int size )
{
   for ( int j = 0; j < size; j++ ) {

      if ( j % 20 == 0 )
         cout << endl;

      cout << setw( 2 ) << a[ j ];
   }
}
```

Fig. 4.17 Survey data analysis program (part 3 of 3).

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788


********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7


********
  Mode
********
Response  Frequency          Histogram

                                      1    1    2    2
                                 5    0    5    0    5

       1          1          *
       2          3          ***
       3          4          ****
       4          5          *****
       5          8          ********
       6          9          *********
       7         23          ***********************
       8         27          ***************************
       9         19          *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Fig. 4.18 Sample run for the survey data analysis program.

```
// Fig. 4.19: fig04_19.cpp
// Linear search of an array
#include <iostream.h>

int linearSearch( const int [], int, int );

int main()
{
   const int arraySize = 100;
   int a[ arraySize ], searchKey, element;

   for ( int x = 0; x < arraySize; x++ )  // create some data
      a[ x ] = 2 * x;

   cout << "Enter integer search key:" << endl;
   cin >> searchKey;
   element = linearSearch( a, searchKey, arraySize );

   if ( element != -1 )
      cout << "Found value in element " << element << endl;
   else
      cout << "Value not found" << endl;

   return 0;
}
```

Fig. 4.19 Linear search of an array (part 1 of 2).

```

int linearSearch( const int array[], int key, int sizeOfArray )
{
   for ( int n = 0; n < sizeOfArray; n++ )
      if ( array[ n ] == key )
         return n;

   return -1;
}
```

```
Enter integer search key:
36
Found value in element 18
```

```
Enter integer search key:
37
Value not found
```

Fig. 4.19 Linear search of an array (part 2 of 2).

```
// Fig. 4.20: fig04_20.cpp
// Binary search of an array
#include <iostream.h>
#include <iomanip.h>

int binarySearch( int [], int, int, int, int );
void printHeader( int );
void printRow( int [], int, int, int, int );

int main()
{
   const int arraySize = 15;
   int a[ arraySize ], key, result;

   for ( int i = 0; i < arraySize; i++ )
      a[ i ] = 2 * i;   // place some data in array

   cout << "Enter a number between 0 and 28: ";
   cin >> key;

   printHeader( arraySize );
   result = binarySearch( a, key, 0, arraySize - 1, arraySize );

```

Fig. 4.20 Binary search of a sorted array (part 1 of 4).

```
   if ( result != -1 )
      cout << '\n' << key << " found in array element "
           << result << endl;
   else
      cout << '\n' << key << " not found" << endl;

   return 0;
}

// Binary search
int binarySearch( int b[], int searchKey, int low, int high,
                  int size )
{
   int middle;

   while ( low <= high ) {
      middle = ( low + high ) / 2;

      printRow( b, low, middle, high, size );

      if ( searchKey == b[ middle ] )  // match
         return middle;
      else if ( searchKey < b[ middle ] )
         high = middle - 1;        // search low end of array
      else
         low = middle + 1;         // search high end of array
   }

   return -1;   // searchKey not found
}

// Print a header for the output
void printHeader( int size )
{
   cout << "\nSubscripts:\n";
   for ( int i = 0; i < size; i++ )
```

```
      cout << setw( 3 ) << i << ' ';

   cout << '\n';

   for ( i = 1; i <= 4 * size; i++ )
      cout << '-';

   cout << endl;
}

```

Fig. 4.20 Binary search of a sorted array (part 2 of 4).

```
// Print one row of output showing the current
// part of the array being processed.
void printRow( int b[], int low, int mid, int high, int size )
{
   for ( int i = 0; i < size; i++ )
      if ( i < low || i > high )
         cout << "    ";
      else if ( i == mid )           // mark middle value
         cout << setw( 3 ) << b[ i ] << '*';
      else
         cout << setw( 3 ) << b[ i ] << ' ';

   cout << endl;
}
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

Fig. 4.20 Binary search of a sorted array (part 3 of 4).

```
Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

Fig. 4.20 Binary search of a sorted array (part 4 of 4).

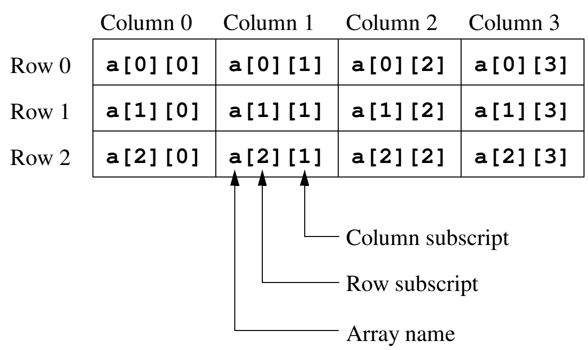


Fig. 4.21 A double-subscripted array with three rows and four columns.

```
// Fig. 4.22: fig04_22.cpp
// Initializing multidimensional arrays
#include <iostream.h>

void printArray( int [][ 3 ] );

int main()
{
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
       array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 },
       array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   cout << "Values in array1 by row are:" << endl;
   printArray( array1 );

   cout << "Values in array2 by row are:" << endl;
   printArray( array2 );

   cout << "Values in array3 by row are:" << endl;
   printArray( array3 );

   return 0;
}

```

Fig. 4.22 Initializing multidimensional arrays (part 1 of 2).

```
void printArray( int a[][ 3 ] )
{
   for ( int i = 0; i < 2; i++ ) {

      for ( int j = 0; j < 3; j++ )
         cout << a[ i ][ j ] << ' ';

      cout << endl;
   }
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 4.22 Initializing multidimensional arrays (part 2 of 2).

```
// Fig. 4.23: fig04_23.cpp
// Double-subscripted array example
#include <iostream.h>
#include <iomanip.h>

const int students = 3;   // number of students
const int exams = 4;      // number of exams

int minimum( int [][ exams ], int, int );
int maximum(int [][ exams ], int, int );
float average( int [], int );
void printArray( int [][ exams ], int, int );

```

Fig. 4.23 Example of using double-subscripted arrays (part 1 of 3).

```
int main()
{
   int studentGrades[ students ][ exams ] =
          { { 77, 68, 86, 73 },
            { 96, 87, 89, 78 },
            { 70, 90, 86, 81 } };

   cout << "The array is:\n";
   printArray( studentGrades, students, exams );
   cout << "\n\nLowest grade: "
        << minimum( studentGrades, students, exams )
        << "\nHighest grade: "
        << maximum( studentGrades, students, exams ) << '\n';

   for ( int person = 0; person < students; person++ )
      cout << "The average grade for student " << person << " is "
           << setiosflags( ios::fixed | ios::showpoint )
           << setprecision( 2 )
           << average( studentGrades[ person ], exams ) << endl;

```

```
   return 0;
}

// Find the minimum grade
int minimum( int grades[][ exams ], int pupils, int tests )
{
   int lowGrade = 100;

   for ( int i = 0; i < pupils; i++ )

      for ( int j = 0; j < tests; j++ )

         if ( grades[ i ][ j ] < lowGrade )
            lowGrade = grades[ i ][ j ];

   return lowGrade;
}

// Find the maximum grade
int maximum( int grades[][ exams ], int pupils, int tests )
{
   int highGrade = 0;

   for ( int i = 0; i < pupils; i++ )

      for ( int j = 0; j < tests; j++ )

```

Fig. 4.23 Example of using double-subscripted arrays (part 2 of 3).

```
         if ( grades[ i ][ j ] > highGrade )
            highGrade = grades[ i ][ j ];

   return highGrade;
}

// Determine the average grade for a particular student
float average( int setOfGrades[], int tests )
{
   int total = 0;

   for ( int i = 0; i < tests; i++ )
      total += setOfGrades[ i ];

   return ( float ) total / tests;
}

// Print the array
void printArray( int grades[][ exams ], int pupils, int tests )
{
   cout << "                 [0]  [1]  [2]  [3]";

   for ( int i = 0; i < pupils; i++ ) {
      cout << "\nstudentGrades[" << i << "] ";

      for ( int j = 0; j < tests; j++ )
         cout << setiosflags( ios::left ) << setw( 5 )
              << grades[ i ][ j ];
   }
}
```

```
The array is:
                  [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Fig. 4.23 Example of using double-subscripted arrays (part 3 of 3).

Illustrations List *(Main Page)*

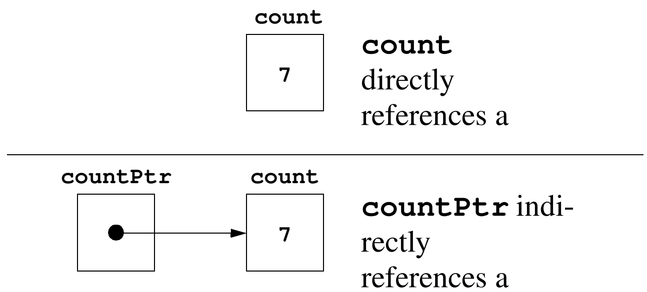


Fig. 5.1 Directly and indirectly referencing a variable.

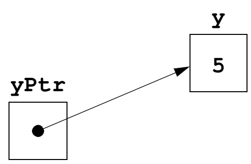


Fig. 5.2 Graphical representation of a pointer pointing to an integer variable in memory.

Fig. 5.3 Representation of `y` and `yPtr` in memory.

```
// Fig. 5.4: fig05_04.cpp
// Using the & and * operators
#include <iostream.h>

int main()
{
   int a;        // a is an integer
   int *aPtr;    // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a;    // aPtr set to address of a

   cout << "The address of a is " << &a
        << "\nThe value of aPtr is " << aPtr;

   cout << "\n\nThe value of a is " << a
        << "\nThe value of *aPtr is " << *aPtr;

   cout << "\n\nShowing that * and & are inverses of "
        << "each other.\n&*aPtr = " << &*aPtr
        << "\n*&aPtr = " << *&aPtr << endl;
   return 0;
}
```

```
The address of a is 0x0064FDF4
The value of aPtr is 0x0064FDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 0x0064FDF4
*&aPtr = 0x0064FDF4
```

Fig. 5.4 The & and * pointer operators.

Operators	Associativity	Type
() []	left to right	highest
++ -- + - ! static_cast<*type*>() & *	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 5.5 Operator precedence and associativity .

```
// Fig. 5.6: fig015_06.cpp
// Cube a variable using call-by-value
#include <iostream.h>

int cubeByValue( int );   // prototype

int main()
{
   int number = 5;

   cout << "The original value of number is " << number;
   number = cubeByValue( number );
   cout << "\nThe new value of number is " << number << endl;
   return 0;
}

int cubeByValue( int n )
{
   return n * n * n;   // cube local variable n
}
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 5.6 Cube a variable using call-by-value.

```
// Fig. 5.7: fig05_07.cpp
// Cube a variable using call-by-reference
// with a pointer argument
#include <iostream.h>

void cubeByReference( int * );   // prototype

int main()
{
   int number = 5;

   cout << "The original value of number is " << number;
   cubeByReference( &number );
   cout << "\nThe new value of number is " << number << endl;
   return 0;
}

void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  // cube number in main
}
```

Fig. 5.7 Cube a variable using call-by-reference with a pointer argument (part 1 of 2).

```
The original value of number is 5
The new value of number is 125
```

Fig. 5.7 Cube a variable using call-by-reference with a pointer argument (part 2 of 2).

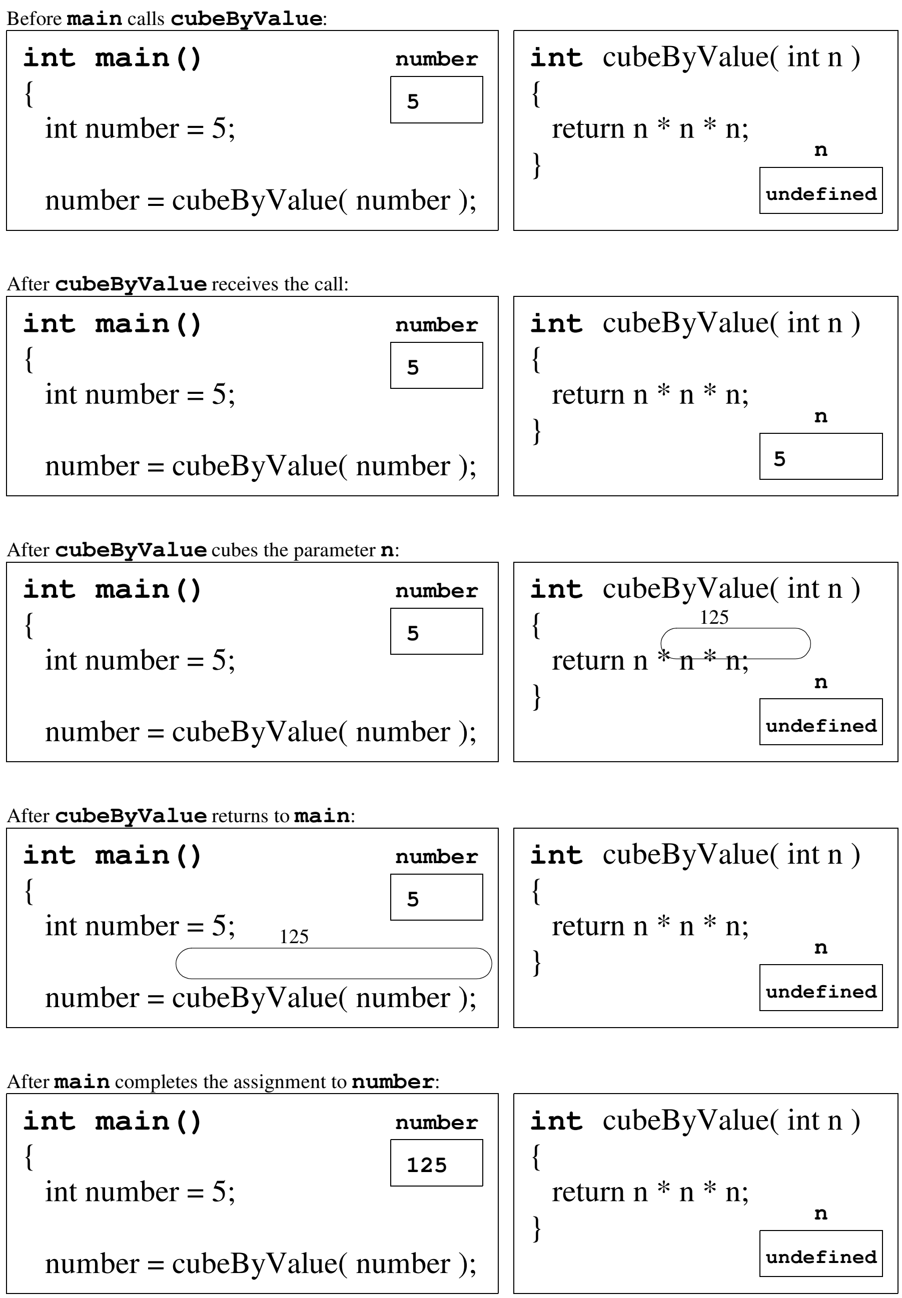


Fig. 5.8 Analysis of a typical call-by-value.

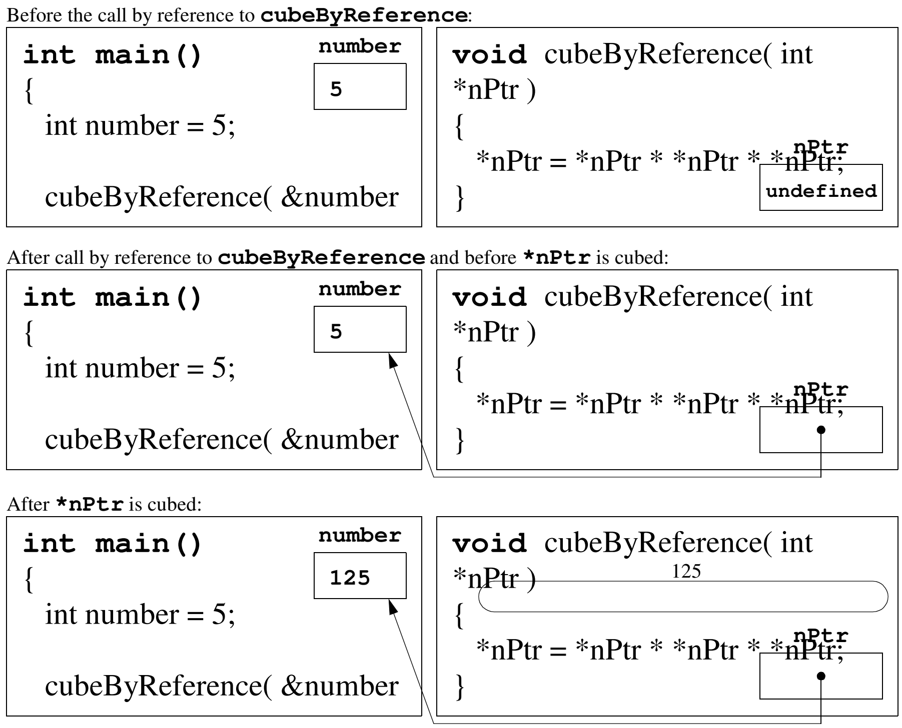


Fig. 5.9 Analysis of a typical call-by-reference with a pointer argument.

```
// Fig. 5.10: fig05_10.cpp
// Converting lowercase letters to uppercase letters
// using a non-constant pointer to non-constant data
#include <iostream.h>
#include <ctype.h>

void convertToUppercase( char * );

int main()
{
   char string[] = "characters and $32.98";

   cout << "The string before conversion is: " << string;
   convertToUppercase( string );
   cout << "\nThe string after conversion is:  "
        << string << endl;
   return 0;
}

void convertToUppercase( char *sPtr )
{
   while ( *sPtr != '\0' ) {

      if ( *sPtr >= 'a' && *sPtr <= 'z' )
         *sPtr = toupper( *sPtr );  // convert to uppercase

      ++sPtr;  // move sPtr to the next character
   }
}
```

```
The string before conversion is: characters and $32.98
The string after conversion is:  CHARACTERS AND $32.98
```

Fig. 5.10 Converting a string to uppercase.

```cpp
// Fig. 5.11: fig05_11.cpp
// Printing a string one character at a time using
// a non-constant pointer to constant data
#include <iostream.h>

void printCharacters( const char * );

int main()
{
   char string[] = "print characters of a string";

   cout << "The string is:\n";
   printCharacters( string );
   cout << endl;
   return 0;
}

// In printCharacters, sPtr is a pointer to a character
// constant. Characters cannot be modified through sPtr
// (i.e., sPtr is a "read-only" pointer).
void printCharacters( const char *sPtr )
{
   for ( ; *sPtr != '\0'; sPtr++ )   // no initialization
      cout << *sPtr;
}
```

```
The string is:
print characters of a string
```

Fig. 5.11 Printing a string one character at a time using a non-constant pointer to constant data.

```
// Fig. 5.12: fig05_12.cpp
// Attempting to modify data through a
// non-constant pointer to constant data.
#include <iostream.h>

void f( const int * );

int main()
{
   int y;
```

Fig. 5.12 Attempting to modify data through a non-constant pointer to constant data (part 1 of 2).

```

   f( &y );     // f attempts illegal modification

   return 0;
}

// In f, xPtr is a pointer to an integer constant
void f( const int *xPtr )
{
   *xPtr = 100;  // cannot modify a const object
}
```

```
Compiling FIG05_12.CPP:
Error FIG05_12.CPP 20: Cannot modify a const object
Warning FIG05_12.CPP 21: Parameter 'xPtr' is never used
```

Fig. 5.12 Attempting to modify data through a non-constant pointer to constant data (part 2 of 2).

```
// Fig. 5.13: fig05_13.cpp
// Attempting to modify a constant pointer to
// non-constant data
#include <iostream.h>

int main()
{
   int x, y;

   int * const ptr = &x; // ptr is a constant pointer to an
                         // integer. An integer can be modified
                         // through ptr, but ptr always points
                         // to the same memory location.
   *ptr = 7;
   ptr = &y;

   return 0;
}
```

```
Compiling FIG05_13.CPP:
Error FIG05_13.CPP 15: Cannot modify a const object
Warning FIG05_13.CPP 18: 'y' is declared but never used
```

Fig. 5.13 Attempting to modify a constant pointer to non-constant data.

```
1  // Fig. 5.14: fig05_14.cpp
2  // Attempting to modify a constant pointer to
3  // constant data.
4  #include <iostream.h>
5
6  int main()
7  {
8     int x = 5, y;
9
10    const int *const ptr = &x; // ptr is a constant pointer to a
11                               // constant integer. ptr always
12                               // points to the same location
13                               // and the integer at that
14                               // location cannot be modified.
15    cout << *ptr << endl;
16    *ptr = 7;
17    ptr = &y;
18
19    return 0;
20 }
```

```
Compiling FIG05_14.CPP:
Error FIG05_14.CPP 16: Cannot modify a const object
Error FIG05_14.CPP 17: Cannot modify a const object
Warning FIG05_14.CPP 20: 'y' is declared but never used
```

Fig. 5.14 Attempting to modify a constant pointer to constant data.

```
// Fig. 5.15: fig05_15.cpp
// This program puts values into an array, sorts the values into
// ascending order, and prints the resulting array.
#include <iostream.h>
#include <iomanip.h>

void bubbleSort( int *, const int );

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int i;

   cout << "Data items in original order\n";

   for ( i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << a[ i ];

   bubbleSort( a, arraySize );          // sort the array
   cout << "\nData items in ascending order\n";

   for ( i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << a[ i ];

   cout << endl;
   return 0;
}

```

```
void bubbleSort( int *array, const int size )
{
   void swap( int *, int * );

   for ( int pass = 0; pass < size - 1; pass++ )

      for ( int j = 0; j < size - 1; j++ )

         if ( array[ j ] > array[ j + 1 ] )
            swap( &array[ j ], &array[ j + 1 ] );
}

void swap( int *element1Ptr, int *element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}
```

Fig. 5.15 Bubble sort with call-by-reference (part 1 of 2).

```
hold = array[ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = hold;
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 5.15 Bubble sort with call-by-reference (part 2 of 2).

```
// Fig. 5.16: fig05_16.cpp
// Sizeof operator when used on an array name
// returns the number of bytes in the array.
#include <iostream.h>

size_t getSize( float * );

int main()
{
   float array[ 20 ];

   cout << "The number of bytes in the array is "
        << sizeof( array )
        << "\nThe number of bytes returned by getSize is "
        << getSize( array ) << endl;

   return 0;
}

size_t getSize( float *ptr )
{
   return sizeof( ptr );
}
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

Fig. 5.16 The **sizeof** operator when applied to an array name returns the number of bytes in the array.

```
// Fig. 5.17: fig05_17.cpp
// Demonstrating the sizeof operator
#include <iostream.h>
#include <iomanip.h>

int main()
{
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int array[ 20 ], *ptr = array;

   cout << "sizeof c = " << sizeof c
        << "\tsizeof(char) = " << sizeof( char )
        << "\nsizeof s = " << sizeof s
        << "\tsizeof(short) = " << sizeof( short )
        << "\nsizeof i = " << sizeof i
        << "\tsizeof(int) = " << sizeof( int )
        << "\nsizeof l = " << sizeof l
        << "\tsizeof(long) = " << sizeof( long )
        << "\nsizeof f = " << sizeof f
        << "\tsizeof(float) = " << sizeof( float )
```

Fig. 5.17 Using the **sizeof** operator to determine standard data type sizes (part 1 of 2).

```
        << "\nsizeof d = " << sizeof d
        << "\tsizeof(double) = " << sizeof( double )
        << "\nsizeof ld = " << sizeof ld
        << "\tsizeof(long double) = " << sizeof( long double )
        << "\nsizeof array = " << sizeof array
        << "\nsizeof ptr = " << sizeof ptr
        << endl;
   return 0;
}
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Fig. 5.17 Using the **sizeof** operator to determine standard data type sizes (part 2 of 2).

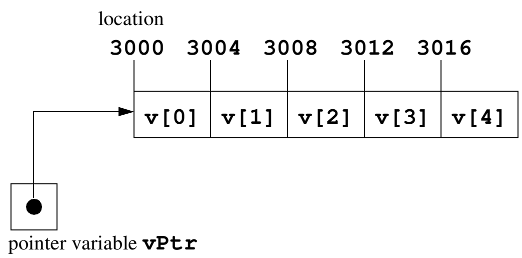


Fig. 5.18 The array **v** and a pointer variable **vPtr** that points to **v**.

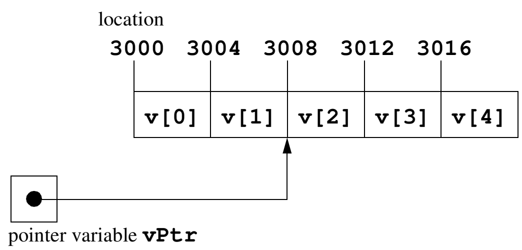


Fig. 5.19 The pointer **vPtr** after pointer arithmetic.

```
// Fig. 5.20: fig05_20.cpp
// Using subscripting and pointer notations with arrays

#include <iostream.h>

int main()
{
   int b[] = { 10, 20, 30, 40 };
   int *bPtr = b;   // set bPtr to point to array b

   cout << "Array b printed with:\n"
        << "Array subscript notation\n";

   for ( int i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';


   cout << "\nPointer/offset notation where\n"
        << "the pointer is the array name\n";

   for ( int offset = 0; offset < 4; offset++ )
      cout << "*(b + " << offset << ") = "
           << *( b + offset ) << '\n';


   cout << "\nPointer subscript notation\n";

   for ( i = 0; i < 4; i++ )
      cout << "bPtr[" << i << "] = " << bPtr[ i ] << '\n';

   cout << "\nPointer/offset notation\n";

   for ( offset = 0; offset < 4; offset++ )
      cout << "*(bPtr + " << offset << ") = "
           << *( bPtr + offset ) << '\n';

   return 0;
}
```

Fig. 5.20 Using four methods of referencing array elements (part 1 of 2).

```
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 5.20 Using four methods of referencing array elements (part 2 of 2).

```
// Fig. 5.21: fig05_21.cpp
// Copying a string using array notation
// and pointer notation.
#include <iostream.h>

void copy1( char *, const char * );
void copy2( char *, const char * );

int main()
{
   char string1[ 10 ], *string2 = "Hello",
        string3[ 10 ], string4[] = "Good Bye";

   copy1( string1, string2 );
   cout << "string1 = " << string1 << endl;

   copy2( string3, string4 );
   cout << "string3 = " << string3 << endl;

   return 0;
}
```

Fig. 5.21 Copying a string using array notation and pointer notation (part 1 of 2).

```
// copy s2 to s1 using array notation
void copy1( char *s1, const char *s2 )
{
   for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
      ;   // do nothing in body
}

// copy s2 to s1 using pointer notation
void copy2( char *s1, const char *s2 )
{
   for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
      ;   // do nothing in body
}
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 5.21 Copying a string using array notation and pointer notation (part 2 of 2).

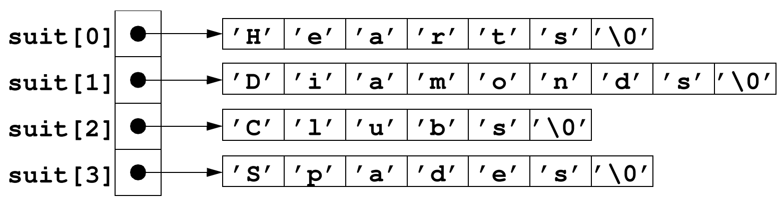


Fig. 5.22 A graphical representation of the **suit** array.

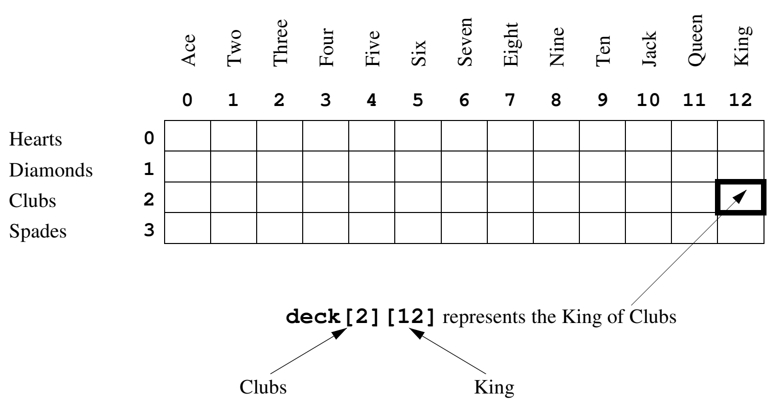


Fig. 5.23 Double-subscripted array representation of a deck of cards.

```
// Fig. 5.24: fig05_24.cpp
// Card shuffling and dealing program
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

void shuffle( int [][ 13 ] );
void deal( const int [][ 13 ], const char *[], const char *[] );

int main()
{
   const char *suit[ 4 ] =
      { "Hearts", "Diamonds", "Clubs", "Spades" };
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };
   int deck[ 4 ][ 13 ] = { 0 };

   srand( time( 0 ) );

   shuffle( deck );
   deal( deck, face, suit );

   return 0;
}

void shuffle( int wDeck[][ 13 ] )
{
   int row, column;

   for ( int card = 1; card <= 52; card++ ) {
      do {
         row = rand() % 4;
         column = rand() % 13;
      } while( wDeck[ row ][ column ] != 0 );

      wDeck[ row ][ column ] = card;
   }
}

void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] )
{
   for ( int card = 1; card <= 52; card++ )

      for ( int row = 0; row <= 3; row++ )

         for ( int column = 0; column <= 12; column++ )

            if ( wDeck[ row ][ column ] == card )
               cout << setw( 5 ) << setiosflags( ios::right )
                    << wFace[ column ] << " of "
                    << setw( 8 ) << setiosflags( ios::left )
                    << wSuit[ row ]
                    << ( card % 2 == 0 ? '\n' : '\t' );
}
```

Fig. 5.24 Card shuffling and dealing program.

```
   Six of Clubs          Seven of Diamonds
   Ace of Spades           Ace of Diamonds
   Ace of Hearts         Queen of Diamonds
 Queen of Clubs          Seven of Hearts
   Ten of Hearts         Deuce of Clubs
   Ten of Spades         Three of Spades
   Ten of Diamonds        Four of Spades
  Four of Diamonds         Ten of Clubs
   Six of Diamonds         Six of Spades
 Eight of Hearts         Three of Diamonds
  Nine of Hearts         Three of Hearts
 Deuce of Spades           Six of Hearts
  Five of Clubs          Eight of Clubs
 Deuce of Diamonds       Eight of Spades
  Five of Spades          King of Clubs
  King of Diamonds        Jack of Spades
 Deuce of Hearts         Queen of Hearts
   Ace of Clubs           King of Spades
 Three of Clubs           King of Hearts
  Nine of Clubs           Nine of Spades
  Four of Hearts         Queen of Spades
 Eight of Diamonds        Nine of Diamonds
  Jack of Diamonds       Seven of Clubs
  Five of Hearts          Five of Diamonds
  Four of Clubs           Jack of Hearts
  Jack of Clubs          Seven of Spades
```

Fig. 5.25 Sample run of card shuffling and dealing program.

```
// Fig. 5.26: fig05_26.cpp
// Multipurpose sorting program using function pointers
#include <iostream.h>
#include <iomanip.h>

void bubble( int [], const int, int (*)( int, int ) );
int ascending( int, int );
int descending( int, int );

int main()
{
   const int arraySize = 10;
   int order,
       counter,
       a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   cout << "Enter 1 to sort in ascending order,\n"
        << "Enter 2 to sort in descending order: ";
   cin >> order;
   cout << "\nData items in original order\n";

   for ( counter = 0; counter < arraySize; counter++ )
      cout << setw( 4 ) << a[ counter ];

   if ( order == 1 ) {
      bubble( a, arraySize, ascending );
      cout << "\nData items in ascending order\n";
   }
```

```
   else {
      bubble( a, arraySize, descending );
      cout << "\nData items in descending order\n";
   }

   for ( counter = 0; counter < arraySize; counter++ )
      cout << setw( 4 ) << a[ counter ];

   cout << endl;
   return 0;
}

void bubble( int work[], const int size,
             int (*compare)( int, int ) )
{
   void swap( int *, int * );

   for ( int pass = 1; pass < size; pass++ )
```

Fig. 5.26 Multipurpose sorting program using function pointers (part 1 of 2).

```

      for ( int count = 0; count < size - 1; count++ )

         if ( (*compare)( work[ count ], work[ count + 1 ] ) )
            swap( &work[ count ], &work[ count + 1 ] );
}

void swap( int *element1Ptr, int *element2Ptr )
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}

int ascending( int a, int b )
{
   return b < a;   // swap if b is less than a
}

int descending( int a, int b )
{
   return b > a;   // swap if b is greater than a
}
```

Fig. 5.26 Multipurpose sorting program using function pointers (part 2 of 2).

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Fig. 5.27 The outputs of the bubble sort program in Fig. 5.26.

```
// Fig. 5.28: fig05_28.cpp
// Demonstrating an array of pointers to functions
#include <iostream.h>
void function1( int );
void function2( int );
void function3( int );

int main()
{
   void (*f[ 3 ])( int ) = { function1, function2, function3 };
   int choice;

   cout << "Enter a number between 0 and 2, 3 to end: ";
   cin >> choice;

   while ( choice >= 0 && choice < 3 ) {
      (*f[ choice ])( choice );
      cout << "Enter a number between 0 and 2, 3 to end: ";
      cin >> choice;
   }

   cout << "Program execution completed." << endl;
   return 0;
}

void function1( int a )
{
   cout << "You entered " << a
        << " so function1 was called\n\n";
}

void function2( int b )
{
   cout << "You entered " << b
        << " so function2 was called\n\n";
}

void function3( int c )
{
   cout << "You entered " << c
        << " so function3 was called\n\n";
}
```

Fig. 5.28 Demonstrating an array of pointers to functions (part 1 of 2).

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed
```

Fig. 5.28 Demonstrating an array of pointers to functions (part 2 of 2).

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 )`	Copies the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strncpy( char *s1, const char *s2, size_t n )`	Copies at most **n** characters of the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strcat( char *s1, const char *s2 )`	Appends the string **s2** to the string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`char *strncat( char *s1, const char *s2, size_t n )`	Appends at most **n** characters of string **s2** to string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`int strcmp( const char *s1, const char *s2 )`	Compares the string **s1** to the string **s2**. The function returns a value of 0, less than 0, or greater than 0 if **s1** is equal to, less than, or greater than **s2**, respectively.
`int strncmp( const char *s1, const char *s2, size_t n )`	Compares up to **n** characters of the string **s1** to the string **s2**. The function returns 0, less than 0, or greater than 0 if **s1** is equal to, less than, or greater than **s2**, respectively.
`char *strtok( char *s1, const char *s2 )`	A sequence of calls to **strtok** breaks string **s1** into "tokens"—logical pieces such as words in a line of text—separated by characters contained in string **s2**. The first call contains **s1** as the first argument, and subsequent calls to continue tokenizing the same string contain **NULL** as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, **NULL** is returned.
`size_t strlen( const char *s )`	Determines the length of string **s**. The number of characters preceding the terminating null character is returned.

Fig. 5.29 The string manipulation functions of the string handling library.

```
// Fig. 5.30: fig05_30.cpp
// Using strcpy and strncpy
#include <iostream.h>
#include <string.h>

int main()
{
   char x[] = "Happy Birthday to You";
   char y[ 25 ], z[ 15 ];

   cout << "The string in array x is: " << x
        << "\nThe string in array y is: " << strcpy( y, x )
        << '\n';
   strncpy( z, x, 14 );  // does not copy null character
   z[ 14 ] = '\0';
   cout << "The string in array z is: " << z << endl;

   return 0;
}
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Fig. 5.30 Using `strcpy` and `strncpy`.

```
// Fig. 5.31: fig05_31.cpp
// Using strcat and strncat
#include <iostream.h>
#include <string.h>

int main()
{
   char s1[ 20 ] = "Happy ";
   char s2[] = "New Year ";
   char s3[ 40 ] = "";

   cout << "s1 = " << s1 << "\ns2 = " << s2;
   cout << "\nstrcat(s1, s2) = " << strcat( s1, s2 );
   cout << "\nstrncat(s3, s1, 6) = " << strncat( s3, s1, 6 );
   cout << "\nstrcat(s3, s1) = " << strcat( s3, s1 ) << endl;

   return 0;
}
```

```
s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
```

Fig. 5.31 Using `strcat` and `strncat`.

```
// Fig. 5.32: fig05_32.cpp
// Using strcmp and strncmp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

int main()
{
   char *s1 = "Happy New Year";
   char *s2 = "Happy New Year";
   char *s3 = "Happy Holidays";

   cout << "s1 = " << s1 << "\ns2 = " << s2
        << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
        << setw( 2 ) << strcmp( s1, s2 )
        << "\nstrcmp(s1, s3) = " << setw( 2 )
        << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
        << setw( 2 ) << strcmp( s3, s1 );

   cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
        << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
        << setw( 2 ) << strncmp( s1, s3, 7 )
        << "\nstrncmp(s3, s1, 7) = "
        << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
   return 0;
}
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Fig. 5.32 Using `strcmp` and `strncmp`.

```
// Fig. 5.33: fig05_33.cpp
// Using strtok
#include <iostream.h>
#include <string.h>

int main()
{
   char string[] = "This is a sentence with 7 tokens";
   char *tokenPtr;

   cout << "The string to be tokenized is:\n" << string
        << "\n\nThe tokens are:\n";

   tokenPtr = strtok( string, " " );

   while ( tokenPtr != NULL ) {
```

```
        cout << tokenPtr << '\n';
        tokenPtr = strtok( NULL, " " );
    }

    return 0;
}
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Fig. 5.33 Using `strtok`.

```
// Fig. 5.34: fig05_34.cpp
// Using strlen
#include <iostream.h>
#include <string.h>

int main()
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "four";
   char *string3 = "Boston";

   cout  << "The length of \"" << string1
         << "\" is " << strlen( string1 )
         << "\nThe length of \"" << string2
         << "\" is " << strlen( string2 )
         << "\nThe length of \"" << string3
         << "\" is " << strlen( string3 ) << endl;

    return 0;
}
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Fig. 5.34 Using `strlen`.

Illustrations List *(Main Page)*

```
// Fig. 6.1: fig06_01.cpp
// Create a structure, set its members, and print it.
#include <iostream.h>

struct Time {    // structure definition
   int hour;     // 0-23
   int minute;   // 0-59
   int second;   // 0-59
};

void printMilitary( const Time & );  // prototype
void printStandard( const Time & );  // prototype

int main()
{
   Time dinnerTime;    // variable of new type Time

   // set members to valid values
   dinnerTime.hour = 18;
   dinnerTime.minute = 30;
   dinnerTime.second = 0;
```

Fig. 6.1 Creating a structure, setting its members, and printing the structure (part 1 of 2).

```

   cout << "Dinner will be held at ";
   printMilitary( dinnerTime );
   cout << " military time,\nwhich is ";
   printStandard( dinnerTime );
   cout << " standard time.\n";

   // set members to invalid values
   dinnerTime.hour = 29;
   dinnerTime.minute = 73;

   cout << "\nTime with invalid values: ";
   printMilitary( dinnerTime );
   cout << endl;
   return 0;
}

// Print the time in military format
void printMilitary( const Time &t )
{
   cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
        << ( t.minute < 10 ? "0" : "" ) << t.minute;
}

// Print the time in standard format
void printStandard( const Time &t )
{
   cout << ( ( t.hour == 0 || t.hour == 12 ) ?
             12 : t.hour % 12 )
        << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
        << ":" << ( t.second < 10 ? "0" : "" ) << t.second
        << ( t.hour < 12 ? " AM" : " PM" );
}
```

```
Dinner will be held at 18:30 military time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73
```

Fig. 6.1 Creating a structure, setting its members, and printing the structure (part 2 of 2).

```
class Time {
public:
   Time();
   void setTime( int, int, int );
   void printMilitary();
   void printStandard();
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};
```

Fig. 6.2 Simple definition of **class Time**.

```
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream.h>

// Time abstract data type (ADT) definition
class Time {
public:
   Time();                          // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();            // print military time format
   void printStandard();            // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }
```

Fig. 6.3 Abstract data type **Time** implementation as a class (part 1 of 3).

```

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Print Time in military format
```

```
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}

// Driver to test simple class Time
int main()
{
   Time t;  // instantiate object t of class Time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime( 13, 27, 6 );
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime( 99, 99, 99 );  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:"
        << "\nMilitary time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

Fig. 6.3 Abstract data type **Time** implementation as a class (part 2 of 3).

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

Fig. 6.3 Abstract data type **Time** implementation as a class (part 3 of 3).

```cpp
// Fig. 6.4: fig06_04.cpp
// Demonstrating the class member access operators . and ->
//
// CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
#include <iostream.h>

// Simple class Count
class Count {
public:
   int x;
   void print() { cout << x << endl; }
};

int main()
{
   Count counter,                // create counter object
         *counterPtr = &counter, // pointer to counter
         &counterRef = counter;  // reference to counter

   cout << "Assign 7 to x and print using the object's name: ";
   counter.x = 7;       // assign 7 to data member x
   counter.print();     // call member function print

   cout << "Assign 8 to x and print using a reference: ";
   counterRef.x = 8;    // assign 8 to data member x
   counterRef.print();  // call member function print

   cout << "Assign 10 to x and print using a pointer: ";
   counterPtr->x = 10;  // assign 10 to data member x
   counterPtr->print(); // call member function print
   return 0;
}
```

```
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10
```

Fig. 6.4 Accessing an object's data members and member functions through each type of object handle—through the object's name, through a reference, and through a pointer to the object.

```
// Fig. 6.5: time1.h
// Declaration of the Time class.
// Member functions are defined in time1.cpp

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {
public:
   Time();                          // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();            // print military time format
   void printStandard();            // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

#endif
```

Fig. 6.5 Separating **Time** class interface and implementation (part 1 of 5).

```
// Fig. 6.5: time1.cpp
// Member function definitions for Time class.
#include <iostream.h>
#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour   = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

```

Fig. 6.5 Separating **Time** class interface and implementation (part 2 of 5).

```
// Print time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 6.5 Separating **Time** class interface and implementation (part 3 of 5).

```
// Fig. 6.5: fig06_05.cpp
// Driver for Time1 class
// NOTE: Compile with time1.cpp
#include <iostream.h>
#include "time1.h"

// Driver to test simple class Time
int main()
{
   Time t;  // instantiate object t of class time

   cout << "The initial military time is ";
   t.printMilitary();
   cout << "\nThe initial standard time is ";
   t.printStandard();

   t.setTime( 13, 27, 6 );
   cout << "\n\nMilitary time after setTime is ";
   t.printMilitary();
   cout << "\nStandard time after setTime is ";
   t.printStandard();

   t.setTime( 99, 99, 99 );  // attempt invalid settings
   cout << "\n\nAfter attempting invalid settings:\n"
        << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();
   cout << endl;
   return 0;
}
```

Fig. 6.5 Separating **Time** class interface and implementation (part 4 of 5).

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM
Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

Fig. 6.5 Separating **Time** class interface and implementation (part 5 of 5).

```
// Fig. 6.6: fig06_06.cpp
// Demonstrate errors resulting from attempts
// to access private class members.
#include <iostream.h>
#include "time1.h"

int main()
{
   Time t;

   // Error: 'Time::hour' is not accessible
   t.hour = 7;

   // Error: 'Time::minute' is not accessible
   cout << "minute = " << t.minute;

   return 0;
}
```

```
Compiling FIG06_06.CPP:
Error FIG06_06.CPP 12: 'Time::hour' is not accessible
Error FIG06_06.CPP 15: 'Time::minute' is not accessible
```

Fig. 6.6 Erroneous attempt to access `private` members of a class.

```
// Fig. 6.7: salesp.h
// SalesPerson class definition
// Member functions defined in salesp.cpp
#ifndef SALESP_H
#define SALESP_H

class SalesPerson {
public:
   SalesPerson();                  // constructor
   void getSalesFromUser(); // get sales figures from keyboard
   void setSales( int, double ); // User supplies one month's
                                 // sales figures.
   void printAnnualSales();

private:
   double totalAnnualSales(); // utility function
   double sales[ 12 ];           // 12 monthly sales figures
};

#endif
```

Fig. 6.7 Using a utility function (part 1 of 5).

```
// Fig. 6.7: salesp.cpp
// Member functions for class SalesPerson
#include <iostream.h>
#include <iomanip.h>
#include "salesp.h"

// Constructor function initializes array
SalesPerson::SalesPerson()
{
   for ( int i = 0; i < 12; i++ )
      sales[ i ] = 0.0;
}

// Function to get 12 sales figures from the user
// at the keyboard
void SalesPerson::getSalesFromUser()
{
   double salesFigure;

   for ( int i = 0; i < 12; i++ ) {
      cout << "Enter sales amount for month "
           << i + 1 << ": ";
      cin >> salesFigure;
      setSales( i, salesFigure );
   }
}

// Function to set one of the 12 monthly sales figures.
// Note that the month value must be from 0 to 11.
void SalesPerson::setSales( int month, double amount )
{
   if ( month >= 0 && month < 12 && amount > 0 )
      sales[ month ] = amount;
   else
      cout << "Invalid month or sales figure" << endl;
}

```

Fig. 6.7 Using a utility function (part 2 of 5).

```
// Print the total annual sales
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 )
        << setiosflags( ios::fixed | ios::showpoint )
        << "\nThe total annual sales are: $"
        << totalAnnualSales() << endl;
}

// Private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0;

   for ( int i = 0; i < 12; i++ )
      total += sales[ i ];

   return total;
}
```

Fig. 6.7 Using a utility function (part 3 of 5).

```
// Fig. 6.7: fig06_07.cpp
// Demonstrating a utility function
// Compile with salesp.cpp
#include "salesp.h"

int main()
{
   SalesPerson s;         // create SalesPerson object s

   s.getSalesFromUser();  // note simple sequential code
   s.printAnnualSales();  // no control structures in main
   return 0;
}
```

Fig. 6.7 Using a utility function (part 4 of 5).

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.58
```

Fig. 6.7 Using a utility function (part 5 of 5).

```
// Fig. 6.8: time2.h
// Declaration of the Time class.
// Member functions are defined in time2.cpp

// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME2_H
#define TIME2_H

// Time abstract data type definition
class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

#endif
```

Fig. 6.8 Using a constructor with default arguments (part 1 of 6).

```
// Fig. 6.8: time2.cpp
// Member function definitions for Time class.
#include <iostream.h>
#include "time2.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

```

Fig. 6.8 Using a constructor with default arguments (part 2 of 6).

```
// Set a new Time value using military time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour   = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 6.8 Using a constructor with default arguments (part 3 of 6).

```
// Fig. 6.8: fig06_08.cpp
// Demonstrating a default constructor
// function for class Time.
#include <iostream.h>
#include "time2.h"

int main()
{
   Time t1,             // all arguments defaulted
        t2(2),          // minute and second defaulted
        t3(21, 34),     // second defaulted
        t4(12, 25, 42), // all values specified
        t5(27, 74, 99); // all bad values specified

   cout << "Constructed with:\n"
        << "all arguments defaulted:\n   ";
   t1.printMilitary();
   cout << "\n   ";
   t1.printStandard();

   cout << "\nhour specified; minute and second defaulted:"
```

```
          << "\n   ";
```

Fig. 6.8 Using a constructor with default arguments (part 4 of 6).

```
     t2.printMilitary();
     cout << "\n   ";
     t2.printStandard();

     cout << "\nhour and minute specified; second defaulted:"
          << "\n   ";
     t3.printMilitary();
     cout << "\n   ";
     t3.printStandard();

     cout << "\nhour, minute, and second specified:"
          << "\n   ";
     t4.printMilitary();
     cout << "\n   ";
     t4.printStandard();

     cout << "\nall invalid values specified:"
          << "\n   ";
     t5.printMilitary();
     cout << "\n   ";
     t5.printStandard();
     cout << endl;

     return 0;
}
```

Fig. 6.8 Using a constructor with default arguments (part 5 of 6).

```
Constructed with:
all arguments defaulted:
   00:00
   12:00:00 AM
hour specified; minute and second defaulted:
   02:00
   2:00:00 AM
hour and minute specified; second defaulted:
   21:34
   9:34:00 PM
hour, minute, and second specified:
   12:25
   12:25:42 PM
all invalid values specified:
   00:00
   12:00:00 AM
```

Fig. 6.8 Using a constructor with default arguments (part 6 of 6).

```cpp
// Fig. 6.9: create.h
// Definition of class CreateAndDestroy.
// Member functions defined in create.cpp.
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};

#endif
```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 1 of 4).

```cpp
// Fig. 6.9: create.cpp
// Member function definitions for class CreateAndDestroy
#include <iostream.h>
#include "create.h"

CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 2 of 4).

```cpp
// Fig. 6.9: fig06_09.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream.h>
#include "create.h"

void create( void );   // prototype

CreateAndDestroy first( 1 );  // global object

int main()
{
   cout << "   (global created before main)" << endl;
```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 3 of 4).

```cpp

   CreateAndDestroy second( 2 );        // local object
   cout << "   (local automatic in main)" << endl;

   static CreateAndDestroy third( 3 );  // local object
   cout << "   (local static in main)" << endl;

   create();  // call function to create objects

   CreateAndDestroy fourth( 4 );        // local object
   cout << "   (local automatic in main)" << endl;
   return 0;
}
```

```
// Function to create objects
void create( void )
{
   CreateAndDestroy fifth( 5 );
   cout << "   (local automatic in create)" << endl;

   static CreateAndDestroy sixth( 6 );
   cout << "   (local static in create)" << endl;

   CreateAndDestroy seventh( 7 );
   cout << "   (local automatic in create)" << endl;
}
```

```
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 4 of 4).

```
// Fig. 6.10: time3.h
// Declaration of the Time class.
// Member functions defined in time3.cpp

// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME3_H
#define TIME3_H

class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // constructor

   // set functions
   void setTime( int, int, int );  // set hour, minute, second
   void setHour( int );    // set hour
   void setMinute( int ); // set minute
   void setSecond( int ); // set second
```

Fig. 6.10 Using set and get functions (part 1 of 6).

```

   // get functions
   int getHour();         // return hour
   int getMinute();       // return minute
   int getSecond();       // return second

   void printMilitary();  // output military time
   void printStandard();  // output standard time

private:
   int hour;              // 0 - 23
   int minute;            // 0 - 59
   int second;            // 0 - 59
};

#endif
```

Fig. 6.10 Using set and get functions (part 2 of 6).

```
// Fig. 6.10: time3.cpp
// Member function definitions for Time class.
#include "time3.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
}

// Set the hour value
void Time::setHour( int h )
   { hour = ( h >= 0 && h < 24 ) ? h : 0; }

// Set the minute value
void Time::setMinute( int m )
   { minute = ( m >= 0 && m < 60 ) ? m : 0; }

// Set the second value
void Time::setSecond( int s )
   { second = ( s >= 0 && s < 60 ) ? s : 0; }

```

Fig. 6.10 Using set and get functions (part 3 of 6).

```
// Get the hour value
int Time::getHour() { return hour; }

// Get the minute value
int Time::getMinute() { return minute; }

// Get the second value
int Time::getSecond() { return second; }

// Print time in military format
```

```
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 6.10 Using set and get functions (part 4 of 6).

```
// Fig. 6.10: fig06_10.cpp
// Demonstrating the Time class set and get functions
#include <iostream.h>
#include "time3.h"

void incrementMinutes( Time &, const int );

int main()
{
   Time t;

   t.setHour( 17 );
   t.setMinute( 34 );
   t.setSecond( 25 );

   cout << "Result of setting all valid values:\n"
        << "  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond();

   t.setHour( 234 );    // invalid hour set to 0
   t.setMinute( 43 );
   t.setSecond( 6373 ); // invalid second set to 0
```

Fig. 6.10 Using set and get functions (part 5 of 6).

```

   cout << "\n\nResult of attempting to set invalid hour and"
        << " second:\n  Hour: " << t.getHour()
        << "  Minute: " << t.getMinute()
        << "  Second: " << t.getSecond() << "\n\n";

   t.setTime( 11, 58, 0 );
   incrementMinutes( t, 3 );

   return 0;
}

void incrementMinutes(Time &tt, const int count)
{
   cout << "Incrementing minute " << count
        << " times:\nStart time: ";
   tt.printStandard();

   for ( int i = 0; i < count; i++ ) {
      tt.setMinute( ( tt.getMinute() + 1 ) % 60);
```

```
      if ( tt.getMinute() == 0 )
         tt.setHour( ( tt.getHour() + 1 ) % 24);

      cout << "\nminute + 1: ";
      tt.printStandard();
   }

   cout << endl;
}
```

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minute: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Fig. 6.10 Using set and get functions (part 6 of 6).

```
// Fig. 6.11: time4.h
// Declaration of the Time class.
// Member functions defined in time4.cpp

// preprocessor directives that
// prevent multiple inclusions of header file
#ifndef TIME4_H
#define TIME4_H

class Time {
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int );  // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
};

#endif
```

Fig. 6.11 Returning a reference to a private data member (part 1 of 4).

```
// Fig. 6.11: time4.cpp
// Member function definitions for Time class.
#include "time4.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
   hour   = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;

   return hour;  // DANGEROUS reference return
}
```

Fig. 6.11 Returning a reference to a private data member (part 2 of 4).

```
// Fig. 6.11: fig06_11.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
// Time class has been trimmed for this example.
#include <iostream.h>
#include "time4.h"

int main()
{
   Time t;
   int &hourRef = t.badSetHour( 20 );

   cout << "Hour before modification: " << hourRef;
   hourRef = 30;  // modification with invalid value
   cout << "\nHour after modification: " << t.getHour();

```

Fig. 6.11 Returning a reference to a private data member (part 3 of 4).

```
   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour(12) = 74;
   cout << "\n\n*********************************\n"
        << "POOR PROGRAMMING PRACTICE!!!!!!!!!\n"
        << "badSetHour as an lvalue, Hour: "
        << t.getHour()
        << "\n*********************************" << endl;

   return 0;
}
```

```
Hour before modification: 20
Hour after modification: 30

*********************************
POOR PROGRAMMING PRACTICE!!!!!!!!
badSetHour as an lvalue, Hour: 74
*********************************
```

Fig. 6.11 Returning a reference to a **`private`** data member (part 4 of 4).

Illustrations List *(Main Page)*

```
// Fig. 7.1: time5.h
// Declaration of the class Time.
// Member functions defined in time5.cpp
#ifndef TIME5_H
#define TIME5_H

class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor

   // set functions
   void setTime( int, int, int );  // set time
   void setHour( int );     // set hour
   void setMinute( int );   // set minute
   void setSecond( int );   // set second

   // get functions (normally declared const)
   int getHour() const;     // return hour
   int getMinute() const;   // return minute
   int getSecond() const;   // return second

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard();        // print standard time
```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 1 of 6).

```
private:
   int hour;               // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};

#endif
```

Fig. 7.1 Using a **Time** class with **const** objects and **const** member functions (part 2 of 6).

```
// Fig. 7.1: time5.cpp
// Member function definitions for Time class.
#include <iostream.h>
#include "time5.h"

// Constructor function to initialize private data.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
void Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
}

// Set the hour value
void Time::setHour( int h )
   { hour = ( h >= 0 && h < 24 ) ? h : 0; }

// Set the minute value
void Time::setMinute( int m )
   { minute = ( m >= 0 && m < 60 ) ? m : 0; }

```

```
// Set the second value
void Time::setSecond( int s )
   { second = ( s >= 0 && s < 60 ) ? s : 0; }

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

// Get the second value
int Time::getSecond() const { return second; }
```

Fig. 7.1 Using a **`Time`** class with **`const`** objects and **`const`** member functions (part 3 of 6).

```

// Display military format time: HH:MM
void Time::printMilitary() const
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard()
{
   cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
        << ( minute < 10 ? "0" : "" ) << minute << ":"
        << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 7.1 Using a **`Time`** class with **`const`** objects and **`const`** member functions (part 4 of 6).

```
// Fig. 7.1: fig07_01.cpp
// Attempting to access a const object with
// non-const member functions.
#include <iostream.h>
#include "time5.h"

int main()
{
   Time wakeUp( 6, 45, 0 );        // non-constant object
   const Time noon( 12, 0, 0 );    // constant object

                              // MEMBER FUNCTION   OBJECT
   wakeUp.setHour( 18 );      // non-const         non-const

   noon.setHour( 12 );        // non-const         const

   wakeUp.getHour();          // const             non-const

   noon.getMinute();          // const             const
   noon.printMilitary();      // const             const
   noon.printStandard();      // non-const         const
   return 0;
}
```

Fig. 7.1 Using a **`Time`** class with **`const`** objects and **`const`** member functions (part 5 of 6).

```
Compiling Fig07_01.cpp
Fig07_01.cpp(15) : error: 'setHour' :
   cannot convert 'this' pointer from
   'const class Time' to 'class Time &'
   Conversion loses qualifiers
Fig07_01.cpp(21) : error: 'printStandard' :
   cannot convert 'this' pointer from
   'const class Time' to 'class Time &'
   Conversion loses qualifiers
```

Fig. 7.1 Using a **`Time`** class with **`const`** objects and **`const`** member functions (part 6 of 6).

```
// Fig. 7.2: fig07_02.cpp
// Using a member initializer to initialize a
// constant of a built-in data type.

#include <iostream.h>

class Increment {
public:
   Increment( int c = 0, int i = 1 );
   void addIncrement() { count += increment; }
   void print() const;

private:
   int count;
   const int increment;// const data member
};

// Constructor for class Increment
Increment::Increment( int c, int i )
   : increment( i )   // initializer for const member
{ count = c; }

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 0; j < 3; j++ ) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Fig. 7.2 Using a member initializer to initialize a constant of a built-in data type.

```
// Fig. 7.3: fig07_03.cpp
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#include <iostream.h>

class Increment {
public:
   Increment( int c = 0, int i = 1 );
   void addIncrement() { count += increment; }
   void print() const;
private:
   int count;
   const int increment;
};

// Constructor for class Increment
Increment::Increment( int c, int i )
{              // Constant member 'increment' is not initialized
   count = c;
   increment = i;  // ERROR: Cannot modify a const object
}

// Print the data
void Increment::print() const
{
   cout << "count = " << count
        << ", increment = " << increment << endl;
}

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 0; j < 3; j++ ) {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   }

   return 0;
}
```

Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment (part 1 of 2).

```
Compiling...
Fig7_3.cpp
Fig7_3.cpp(18) : error: 'increment' :
   must be initialized in constructor base/member
   initializer list
Fig7_3.cpp(20) : error: l-value specifies const object
```

Fig. 7.3 Erroneous attempt to initialize a constant of a built-in data type by assignment (part 2 of 2).

```
// Fig. 7.4: date1.h
// Declaration of the Date class.
// Member functions defined in date1.cpp
#ifndef DATE1_H
#define DATE1_H

class Date {
public:
   Date( int = 1, int = 1, int = 1900 ); // default constructor
   void print() const;  // print date in month/day/year format
   ~Date();  // provided to confirm destruction order
private:
   int month;  // 1-12
   int day;    // 1-31 based on month
   int year;   // any year

   // utility function to test proper day for month and year
   int checkDay( int );
};

#endif
```

Fig. 7.4 Using member-object initializers (part 1 of 6).

```
// Fig. 7.4: date.cpp
// Member function definitions for Date class.
#include <iostream.h>
#include "date1.h"

// Constructor: Confirm proper value for month;
// call utility function checkDay to confirm proper
// value for day.
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= 12 )       // validate the month
      month = mn;
   else {
      month = 1;
      cout << "Month " << mn << " invalid. Set to month 1.\n";
   }

   year = yr;                      // should validate yr
   day = checkDay( dy );           // validate the day

   cout << "Date object constructor for date ";
   print();          // interesting: a print with no arguments
   cout << endl;
}

// Print Date object in form  month/day/year
```

```
void Date::print() const
   { cout << month << '/' << day << '/' << year; }

// Destructor: provided to confirm destruction order
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;
}

```

Fig. 7.4 Using member-object initializers (part 2 of 6).

```
// Utility function to confirm proper day value
// based on month and year.
// Is the year 2000 a leap year?
int Date::checkDay( int testDay )
{
   static const int daysPerMonth[ 13 ] =
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   if ( month == 2 &&      // February: Check for leap year
        testDay == 29 &&
        ( year % 400 == 0 ||                      // year 2000?
         ( year % 4 == 0 && year % 100 != 0 ) ) ) // year 2000?
      return testDay;

   cout << "Day " << testDay << " invalid. Set to day 1.\n";

   return 1;  // leave object in consistent state if bad value
}
```

Fig. 7.4 Using member-object initializers (part 3 of 6).

```
// Fig. 7.4: emply1.h
// Declaration of the Employee class.
// Member functions defined in emply1.cpp
#ifndef EMPLY1_H
#define EMPLY1_H

#include "date1.h"

class Employee {
public:
   Employee( char *, char *, int, int, int, int, int, int );
   void print() const;
   ~Employee();  // provided to confirm destruction order
private:
   char firstName[ 25 ];
   char lastName[ 25 ];
   const Date birthDate;
   const Date hireDate;
};

#endif
```

Fig. 7.4 Using member-object initializers (part 4 of 6).

```
// Fig. 7.4: emply1.cpp
// Member function definitions for Employee class.
#include <iostream.h>
#include <string.h>
#include "emply1.h"
#include "date1.h"

Employee::Employee( char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
   : birthDate( bmonth, bday, byear ),
     hireDate( hmonth, hday, hyear )
{
   // copy fname into firstName and be sure that it fits
   int length = strlen( fname );
   length = ( length < 25 ? length : 24 );
   strncpy( firstName, fname, length );
   firstName[ length ] = '\0';

   // copy lname into lastName and be sure that it fits
   length = strlen( lname );
   length = ( length < 25 ? length : 24 );
   strncpy( lastName, lname, length );
   lastName[ length ] = '\0';

   cout << "Employee object constructor: "
        << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
   cout << lastName << ", " << firstName << "\nHired: ";
   hireDate.print();
   cout << "  Birth date: ";
   birthDate.print();
   cout << endl;
}

// Destructor: provided to confirm destruction order
Employee::~Employee()
{
   cout << "Employee object destructor: "
        << lastName << ", " << firstName << endl;
}
```

Fig. 7.4 Using member-object initializers (part 5 of 6).

```
// Fig. 7.4: fig07_04.cpp
// Demonstrating composition: an object with member objects.
#include <iostream.h>
#include "emply1.h"

int main()
{
   Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );

   cout << '\n';
   e.print();

   cout << "\nTest Date constructor with invalid values:\n";
   Date d( 14, 35, 1994 );  // invalid Date values
   cout << endl;
   return 0;
}
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/1988  Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

Fig. 7.4 Using member-object initializers (part 6 of 6).

```
// Fig. 7.5: fig07_05.cpp
// Friends can access private members of a class.
#include <iostream.h>

// Modified Count class
class Count {
   friend void setX( Count &, int ); // friend declaration
public:
   Count() { x = 0; }                 // constructor
   void print() const { cout << x << endl; }  // output
private:
   int x;  // data member
};

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX( Count &c, int val )
{
   c.x = val;  // legal: setX is a friend of Count
}

int main()
{
   Count counter;

   cout << "counter.x after instantiation: ";
   counter.print();
   cout << "counter.x after call to setX friend function: ";
   setX( counter, 8 );  // set x with a friend
   counter.print();
   return 0;
}
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 7.5 Friends can access `private` members of a class.

```
// Fig. 7.6: fig07_06.cpp
// Non-friend/non-member functions cannot access
// private data of a class.
#include <iostream.h>

// Modified Count class
class Count {
public:
  Count() { x = 0; }                    // constructor
  void print() const { cout << x << endl; }  // output
private:
   int x;   // data member
};

// Function tries to modify private data of Count,
// but cannot because it is not a friend of Count.
void cannotSetX( Count &c, int val )
{
   c.x = val;   // ERROR: 'Count::x' is not accessible
}

int main()
{
   Count counter;

   cannotSetX( counter, 3 ); // cannotSetX is not a friend
   return 0;
}
```

```
Compiling...
Fig07_06.cpp
Fig07_06.cpp(19) : error: 'x' :
   cannot access private member declared in class 'Count'
```

Fig. 7.6 Non-`friend`/non-member functions cannot access `private` class members.

```
// Fig. 7.7: fig07_07.cpp
// Using the this pointer to refer to object members.
#include <iostream.h>

class Test {
public:
   Test( int = 0 );              // default constructor
   void print() const;
private:
   int x;
};

Test::Test( int a ) { x = a; }  // constructor

```

Fig. 7.7 Using the `this` pointer (part 1 of 2).

```
void Test::print() const    // ( ) around *this required
{
   cout << "        x = " << x
        << "\n  this->x = " << this->x
        << "\n(*this).x = " << ( *this ).x << endl;
}

int main()
{
   Test testObject( 12 );

   testObject.print();

   return 0;
}
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 7.7 Using the `this` pointer (part 2 of 2).

```
// Fig. 7.8: time6.h
// Cascading member function calls.

// Declaration of class Time.
// Member functions defined in time6.cpp
#ifndef TIME6_H
#define TIME6_H

class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor

   // set functions
   Time &setTime( int, int, int ); // set hour, minute, second
   Time &setHour( int );     // set hour
   Time &setMinute( int );   // set minute
   Time &setSecond( int );   // set second

   // get functions (normally declared const)
   int getHour() const;      // return hour
   int getMinute() const;    // return minute
   int getSecond() const;    // return second

   // print functions (normally declared const)
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
private:
   int hour;                // 0 - 23
   int minute;              // 0 - 59
   int second;              // 0 - 59
};

#endif
```

Fig. 7.8 Cascading member function calls (part 1 of 4).

```
// Fig. 7.8: time.cpp
// Member function definitions for Time class.
#include "time6.h"
#include <iostream.h>

// Constructor function to initialize private data.
// Calls member function setTime to set variables.
// Default values are 0 (see class definition).
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

// Set the values of hour, minute, and second.
Time &Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this;   // enables cascading
}

// Set the hour value
Time &Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;

   return *this;   // enables cascading
}

// Set the minute value
Time &Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0;

   return *this;   // enables cascading
}

// Set the second value
Time &Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0;

   return *this;   // enables cascading
}

// Get the hour value
int Time::getHour() const { return hour; }

// Get the minute value
int Time::getMinute() const { return minute; }

```

Fig. 7.8 Cascading member function calls (part 2 of 4).

```
// Get the second value
int Time::getSecond() const { return second; }

// Display military format time: HH:MM
void Time::printMilitary() const
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Display standard format time: HH:MM:SS AM (or PM)
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Fig. 7.8 Cascading member function calls (part 3 of 4).

```
// Fig. 7.8: fig07_08.cpp
// Cascading member function calls together
// with the this pointer
#include <iostream.h>
#include "time6.h"

int main()
{
   Time t;

   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
   cout << "Military time: ";
   t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();

   cout << "\n\nNew standard time: ";
   t.setTime( 20, 20, 20 ).printStandard();
   cout << endl;

   return 0;
}
```

```
Military time: 18:30
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 7.8 Cascading member function calls (part 4 of 4).

```
// Fig. 7.9: employ1.h
// An employee class
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
   Employee( const char*, const char* );  // constructor
   ~Employee();                           // destructor
   const char *getFirstName() const;      // return first name
   const char *getLastName() const;       // return last name

   // static member function
   static int getCount();  // return # objects instantiated

private:
   char *firstName;
   char *lastName;

   // static data member
   static int count;  // number of objects instantiated
};

#endif
```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 1 of 5).

```
// Fig. 7.9: employ1.cpp
// Member function definitions for class Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"

// Initialize the static data member
int Employee::count = 0;

// Define the static member function that
// returns the number of employee objects instantiated.
int Employee::getCount() { return count; }

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 );   // ensure memory allocated
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );    // ensure memory allocated
   strcpy( lastName, last );

   ++count;  // increment static count of employees
   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called." << endl;
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;
```

```
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory
   --count;  // decrement static count of employees
}

// Return first name of employee
const char *Employee::getFirstName() const
{
   // const before return type prevents client modifying
   // private data. Client should copy returned string before
   // destructor deletes storage to prevent undefined pointer.
   return firstName;
}
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 2 of 5).

```

// Return last name of employee
const char *Employee::getLastName() const
{
   // const before return type prevents client modifying
   // private data. Client should copy returned string before
   // destructor deletes storage to prevent undefined pointer.
   return lastName;
}
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 3 of 5).

```
// Fig. 7.9: fig07_09.cpp
// Driver to test the Employee class
#include <iostream.h>
#include "employ1.h"

int main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name

   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();

   cout << "\n\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";

   delete e1Ptr;   // recapture memory
   e1Ptr = 0;
   delete e2Ptr;   // recapture memory
   e2Ptr = 0;

   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;
}
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 4 of 5).

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

Fig. 7.9 Using a `static` data member to maintain a count of the number of objects of a class (part 5 of 5).

```
// Fig. 7.10: implementation.h
// Header file for class Implementation

class Implementation {
   public:
      Implementation( int v ) { value = v; }
      void setValue( int v ) { value = v; }
      int getValue() const { return value; }

   private:
      int value;
};
```

Fig. 7.10 Implementing a proxy class (part 1 of 4).

```
// Fig. 7.10: interface.h
// Header file for interface.cpp
class Implementation;    // forward class declaration

class Interface {
   public:
      Interface( int );
      void setValue( int );  // same public interface as
      int getValue() const;  // class Implementation
   private:
      Implementation *ptr;   // requires previous
                             // forward declaration
};
```

Fig. 7.10 Implementing a proxy class (part 2 of 4).

```cpp
// Fig. 7.10: interface.cpp
// Definition of class Interface
#include "interface.h"
#include "implementation.h"

Interface::Interface( int v )
   : ptr ( new Implementation( v ) ) { }

// call Implementation's setValue function
void Interface::setValue( int v ) { ptr->setValue( v ); }

// call Implementation's getValue function
int Interface::getValue() const { return ptr->getValue(); }
```

Fig. 7.10 Implementing a proxy class (part 3 of 4).

```cpp
// Fig. 7.10: fig07_10.cpp
// Hiding a class’s private data with a proxy class.
#include <iostream.h>
#include "interface.h"

int main()
{
   Interface i( 5 );

   cout << "Interface contains: " << i.getValue()
        << " before setValue" << endl;
   i.setValue( 10 );
   cout << "Interface contains: " << i.getValue()
        << " after setValue" << endl;
   return 0;
}
```

```
Interface contains: 5 before setVal
Interface contains: 10 after setVal
```

Fig. 7.10 Implementing a proxy class (part 4 of 4).

Illustrations List *(Main Page)*

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Fig. 8.1 Operators that can be overloaded.

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Fig. 8.2 Operators that cannot be overloaded.

```
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream.h>
#include <iomanip.h>

class PhoneNumber {
   friend ostream &operator<<( ostream&, const PhoneNumber & );
   friend istream &operator>>( istream&, PhoneNumber & );

private:
   char areaCode[ 4 ];  // 3-digit area code and null
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null
};
```

Fig. 8.3 User-defined stream-insertion and stream-extraction operators (part 1 of 2).

```
// Overloaded stream-insertion operator (cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;).
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;
   return output;     // enables cout << a << b << c;
}

istream &operator>>( istream &input, PhoneNumber &num )
{
   input.ignore();                     // skip (
   input >> setw( 4 ) >> num.areaCode; // input area code
   input.ignore( 2 );                  // skip ) and space
   input >> setw( 4 ) >> num.exchange; // input exchange
   input.ignore();                     // skip dash (-)
   input >> setw( 5 ) >> num.line;     // input line
   return input;      // enables cin >> a >> b >> c;
}

int main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (123) 456-7890:\n";

   // cin >> phone invokes operator>> function by
   // issuing the call operator>>( cin, phone ).
   cin >> phone;

   // cout << phone invokes operator<< function by
   // issuing the call operator<<( cout, phone ).
   cout << "The phone number entered was: " << phone << endl;
   return 0;
}
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Fig. 8.3 User-defined stream-insertion and stream-extraction operators (part 2 of 2).

```
// Fig. 8.4: array1.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 );                   // default constructor
   Array( const Array & );              // copy constructor
   ~Array();                            // destructor
   int getSize() const;                 // return size
   const Array &operator=( const Array & ); // assign arrays
   bool operator==( const Array & ) const;  // compare equal

   // Determine if two arrays are not equal and
   // return true, otherwise return false (uses operator==).
   bool operator!=( const Array &right ) const
      { return ! ( *this == right ); }

   int &operator[]( int );              // subscript operator
   const int &operator[]( int ) const;  // subscript operator
   static int getArrayCount();          // Return count of
                                        // arrays instantiated.
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
   static int arrayCount;  // # of Arrays instantiated
};

#endif
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 1 of 8).

```
// Fig 8.4: array1.cpp
// Member function definitions for class Array
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <assert.h>
#include "array1.h"

// Initialize static data member at file scope
int Array::arrayCount = 0;   // no objects yet

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
}

// Copy constructor for class Array
// must receive a reference to prevent infinite recursion
Array::Array( const Array &init ) : size( init.size )
{
```

```
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = init.ptr[ i ];  // copy init into object
}

// Destructor for class Array
Array::~Array()
{
   delete [] ptr;            // reclaim space for array
   --arrayCount;             // one fewer objects
}

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) {  // check for self-assignment

```

Fig. 8.4 Demonstrating an **`Array`** class with overloaded operators (part 2 of 8).

```
      // for arrays of different sizes, deallocate original
      // left side array, then allocate new left side array.
      if ( size != right.size ) {
         delete [] ptr;         // reclaim space
         size = right.size;     // resize this object
         ptr = new int[ size ]; // create space for array copy
         assert( ptr != 0 );    // terminate if not allocated
      }

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ];  // copy array into object
   }

   return *this;   // enables x = y = z;
}

// Determine if two arrays are equal and
// return true, otherwise return false.
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false;    // arrays of different sizes

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true;        // arrays are equal
}

// Overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

```

```
   return ptr[ subscript ]; // reference return
}

// Overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

   return ptr[ subscript ]; // const reference return
}

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 3 of 8).

```
// Return the number of Array objects instantiated
// static functions cannot be const
int Array::getArrayCount() { return arrayCount; }

// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input;   // enables cin >> x >> y;
}

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;
   }

   if ( i % 4 != 0 )
      output << endl;

   return output;   // enables cout << x << y;
}
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 4 of 8).

```
// Fig. 8.4: fig08_04.cpp
// Driver for simple class Array
#include <iostream.h>
#include "array1.h"

int main()
{
   // no objects yet
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';

   // create two arrays and print Array count
   Array integers1( 7 ), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << "\n\n";

```

Fig. 8.4 Demonstrating an **`Array`** class with overloaded operators (part 5 of 8).

```
   // print integers1 size and contents
   cout << "Size of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n"
        << integers1 << '\n';

   // print integers2 size and contents
   cout << "Size of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n"
        << integers2 << '\n';

   // input and print integers1 and integers2
   cout << "Input 17 integers:\n";
   cin >> integers1 >> integers2;
   cout << "After input, the arrays contain:\n"
        << "integers1:\n" << integers1
        << "integers2:\n" << integers2 << '\n';

   // use overloaded inequality (!=) operator
   cout << "Evaluating: integers1 != integers2\n";
   if ( integers1 != integers2 )
      cout << "They are not equal\n";

   // create array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3( integers1 );

   cout << "\nSize of array integers3 is "
        << integers3.getSize()
        << "\nArray after initialization:\n"
        << integers3 << '\n';

   // use overloaded assignment (=) operator
   cout << "Assigning integers2 to integers1:\n";
   integers1 = integers2;
   cout << "integers1:\n" << integers1
        << "integers2:\n" << integers2 << '\n';

   // use overloaded equality (==) operator
   cout << "Evaluating: integers1 == integers2\n";
   if ( integers1 == integers2 )
      cout << "They are equal\n\n";

   // use overloaded subscript operator to create rvalue
```

```
   cout << "integers1[5] is " << integers1[5] << '\n';

   // use overloaded subscript operator to create lvalue
   cout << "Assigning 1000 to integers1[5]\n";
   integers1[5] = 1000;
   cout << "integers1:\n" << integers1 << '\n';
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 6 of 8).

```

   // attempt to use out of range subscript
   cout << "Attempt to assign 1000 to integers1[15]" << endl;
   integers1[15] = 1000;  // ERROR: out of range

   return 0;
}
```

```
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
They are not equal

Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 7 of 8).

```
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
file Array1.cpp, line 87

abnormal program termination
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 8 of 8).

```
// Fig. 8.5: string1.h
// Definition of a String class
#ifndef STRING1_H
#define STRING1_H

#include <iostream.h>

class String {
   friend ostream &operator<<( ostream &, const String & );
   friend istream &operator>>( istream &, String & );

public:
   String( const char * = "" ); // conversion/default ctor
   String( const String & );    // copy constructor
   ~String();                   // destructor
   const String &operator=( const String & );  // assignment
   const String &operator+=( const String & ); // concatenation
   bool operator!() const;                     // is String empty?
   bool operator==( const String & ) const; // test s1 == s2
   bool operator<( const String & ) const;  // test s1 < s2

   // test s1 != s2
   bool operator!=( const String & right ) const
      { return !( *this == right ); }

   // test s1 > s2
   bool operator>( const String &right ) const
      { return right < *this; }

```

Fig. 8.5 Definition of a basic **String** class (part 1 of 9).

```
   // test s1 <= s2
   bool operator<=( const String &right ) const
      { return !( right < *this ); }

   // test s1 >= s2
   bool operator>=( const String &right ) const
      { return !( *this < right ); }

   char &operator[]( int );              // subscript operator
   const char &operator[]( int ) const; // subscript operator
   String &operator()( int, int ); // return a substring
   int getLength() const;          // return string length

private:
   int length;                    // string length
   char *sPtr;                    // pointer to start of string

   void setString( const char * );  // utility function
};

#endif
```

Fig. 8.5 Definition of a basic **String** class (part 2 of 9).

```
// Fig. 8.5: string1.cpp
// Member function definitions for class String
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <assert.h>
#include "string1.h"

// Conversion constructor: Convert char * to String
String::String( const char *s ) : length( strlen( s ) )
{
   cout << "Conversion constructor: " << s << '\n';
   setString( s );         // call utility function
}

// Copy constructor
String::String( const String &copy ) : length( copy.length )
{
   cout << "Copy constructor: " << copy.sPtr << '\n';
   setString( copy.sPtr ); // call utility function
}

// Destructor
String::~String()
{
   cout << "Destructor: " << sPtr << '\n';
```

Fig. 8.5 Definition of a basic **String** class (part 3 of 9).

```
   delete [] sPtr;         // reclaim string
}

// Overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
{
   cout << "operator= called\n";

   if ( &right != this ) {         // avoid self assignment
      delete [] sPtr;              // prevents memory leak
      length = right.length;       // new String length
      setString( right.sPtr );     // call utility function
   }
   else
      cout << "Attempted assignment of a String to itself\n";

   return *this;   // enables cascaded assignments
}

// Concatenate right operand to this object and
// store in this object.
const String &String::operator+=( const String &right )
{
   char *tempPtr = sPtr;        // hold to be able to delete
   length += right.length;      // new String length
   sPtr = new char[ length + 1 ]; // create space
   assert( sPtr != 0 );   // terminate if memory not allocated
   strcpy( sPtr, tempPtr );     // left part of new String
   strcat( sPtr, right.sPtr );  // right part of new String
   delete [] tempPtr;           // reclaim old space
   return *this;                // enables cascaded calls
}

// Is this String empty?
bool String::operator!() const { return length == 0; }

// Is this String equal to right String?
bool String::operator==( const String &right ) const
   { return strcmp( sPtr, right.sPtr ) == 0; }

// Is this String less than right String?
bool String::operator<( const String &right ) const
   { return strcmp( sPtr, right.sPtr ) < 0; }

// Return a reference to a character in a String as an lvalue.
char &String::operator[]( int subscript )
{
   // First test for subscript out of range
   assert( subscript >= 0 && subscript < length );

```

Fig. 8.5 Definition of a basic `String` class (part 4 of 9).

```
   return sPtr[ subscript ];  // creates lvalue
}

// Return a reference to a character in a String as an rvalue.
const char &String::operator[]( int subscript ) const
{
   // First test for subscript out of range
   assert( subscript >= 0 && subscript < length );

   return sPtr[ subscript ];  // creates rvalue
}

// Return a substring beginning at index and
// of length subLength as a reference to a String object.
String &String::operator()( int index, int subLength )
{
   // ensure index is in range and substring length >= 0
   assert( index >= 0 && index < length && subLength >= 0 );

   String *subPtr = new String; // empty String
   assert( subPtr != 0 ); // ensure new String allocated

   // determine length of substring
   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      subPtr->length = length - index + 1;
   else
      subPtr->length = subLength + 1;

   // allocate memory for substring
   delete subPtr->sPtr; // delete character array from object
   subPtr->sPtr = new char[ subPtr->length ];
   assert( subPtr->sPtr != 0 ); // ensure space allocated

   // copy substring into new String
   strncpy( subPtr->sPtr, &sPtr[ index ], subPtr->length );
   subPtr->sPtr[ subPtr->length ] = '\0'; // terminate String

   return *subPtr;          // return new String
}

// Return string length
int String::getLength() const { return length; }

// Utility function to be called by constructors and
// assignment operator.
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate storage
   assert( sPtr != 0 );  // terminate if memory not allocated
   strcpy( sPtr, string2 );       // copy literal to object
}
```

Fig. 8.5 Definition of a basic `String` class (part 5 of 9).

```

// Overloaded output operator
ostream &operator<<( ostream &output, const String &s )
{
   output << s.sPtr;
   return output;   // enables cascading
}

// Overloaded input operator
istream &operator>>( istream &input, String &s )
{
   char temp[ 100 ];   // buffer to store input

   input >> setw( 100 ) >> temp;
   s = temp;        // use String class assignment operator
   return input;    // enables cascading
}
```

Fig. 8.5 Member function definitions for class **String** (part 6 of 9).

```
// Fig. 8.5: fig08_05.cpp
// Driver for class String
#include <iostream.h>
#include "string1.h"

int main()
{
   String s1( "happy" ), s2( " birthday" ), s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
        << "\"; s3 is \"" << s3 << '\"'
        << "\nThe results of comparing s2 and s1:"
        << "\ns2 == s1 yields "
        << ( s2 == s1 ? "true" : "false" )
        << "\ns2 != s1 yields "
        << ( s2 != s1 ? "true" : "false" )
        << "\ns2 >  s1 yields "
        << ( s2 > s1 ? "true" : "false" )
        << "\ns2 <  s1 yields "
        << ( s2 < s1 ? "true" : "false" )
        << "\ns2 >= s1 yields "
        << ( s2 >= s1 ? "true" : "false" )
        << "\ns2 <= s1 yields "
        << ( s2 <= s1 ? "true" : "false" );

   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:\n";
   if ( !s3 ) {
      cout << "s3 is empty; assigning s1 to s3;\n";
```

Fig. 8.5 Definition of a basic **String** class (part 7 of 9).

```
      s3 = s1;                 // test overloaded assignment
      cout << "s3 is \"" << s3 << "\"";
   }

   // test overloaded String concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2;                   // test overloaded concatenation
   cout << s1;

   // test conversion constructor
   cout << "\n\ns1 += \" to you\" yields\n";
   s1 += " to you";            // test conversion constructor
   cout << "s1 = " << s1 << "\n\n";

   // test overloaded function call operator () for substring
   cout << "The substring of s1 starting at\n"
        << "location 0 for 14 characters, s1(0, 14), is:\n"
        << s1( 0, 14 ) << "\n\n";

   // test substring "to-end-of-String" option
   cout << "The substring of s1 starting at\n"
        << "location 15, s1(15, 0), is: "
        << s1( 15, 0 ) << "\n\n";  // 0 is "to end of string"

   // test copy constructor
   String *s4Ptr = new String(s1);
   cout << "*s4Ptr = " << *s4Ptr << "\n\n";

   // test assignment (=) operator with self-assignment
   cout << "assigning *s4Ptr to *s4Ptr\n";
   *s4Ptr = *s4Ptr;             // test overloaded assignment
   cout << "*s4Ptr = " << *s4Ptr << '\n';

   // test destructor
   delete s4Ptr;

   // test using subscript operator to create lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
        << s1 << "\n\n";

   // test subscript out of range
   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
   s1[ 30 ] = 'd';      // ERROR: subscript out of range

   return 0;
}
```

Fig. 8.5 Definition of a basic **String** class (part 8 of 9).

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday
to you

Attempt to assign 'd' to s1[30] yields:
Assertion failed: subscript >= 0 && subscript < length,
file String1.cpp, line 76

abnormal program termination
```

Fig. 8.5 Definition of a basic **`String`** class (part 9 of 9).

```
// Fig. 8.6: date1.h
// Definition of class Date
#ifndef DATE1_H
#define DATE1_H
#include <iostream.h>

class Date {
   friend ostream &operator<<( ostream &, const Date & );

public:
   Date( int m = 1, int d = 1, int y = 1900 ); // constructor
   void setDate( int, int, int ); // set the date
   Date &operator++();            // preincrement operator
   Date operator++( int );        // postincrement operator
   const Date &operator+=( int ); // add days, modify object
   bool leapYear( int );          // is this a leap year?
   bool endOfMonth( int );        // is this end of month?

private:
   int month;
   int day;
   int year;

   static const int days[];       // array of days per month
   void helpIncrement();          // utility function
};

#endif
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 1 of 7).

```
// Fig. 8.6: date1.cpp
// Member function definitions for Date class
#include <iostream.h>
#include "date1.h"

// Initialize static member at file scope;
// one class-wide copy.
const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int m, int d, int y ) { setDate( m, d, y ); }

// Set the date
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

```

Fig. 8.6 Class **Date** with overloaded increment operators (part 2 of 7).

```
   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
}

// Preincrement operator overloaded as a member function.
Date &Date::operator++()
{
   helpIncrement();
   return *this;  // reference return to create an lvalue
}

// Postincrement operator overloaded as a member function.
// Note that the dummy integer parameter does not have a
// parameter name.
Date Date::operator++( int )
{
   Date temp = *this;
   helpIncrement();

   // return non-incremented, saved, temporary object
   return temp;   // value return; not a reference return
}

// Add a specific number of days to a date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this;    // enables cascading
}

// If the year is a leap year, return true;
// otherwise, return false
bool Date::leapYear( int y )
{
   if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
      return true;   // a leap year
   else
      return false;  // not a leap year
}

```

Fig. 8.6 Class **Date** with overloaded increment operators (part 3 of 7).

```
// Determine if the day is the end of the month
bool Date::endOfMonth( int d )
{
   if ( month == 2 && leapYear( year ) )
      return d == 29; // last day of Feb. in leap year
   else
      return d == days[ month ];
}

// Function to help increment the date
void Date::helpIncrement()
{
   if ( endOfMonth( day ) && month == 12 ) {  // end year
      day = 1;
      month = 1;
      ++year;
```

```
   }
   else if ( endOfMonth( day ) ) {            // end month
      day = 1;
      ++month;
   }
   else       // not end of month or year; increment day
      ++day;
}

// Overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static char *monthName[ 13 ] = { "", "January",
      "February", "March", "April", "May", "June",
      "July", "August", "September", "October",
      "November", "December" };

   output << monthName[ d.month ] << ' '
          << d.day << ", " << d.year;

   return output;   // enables cascading
}
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 4 of 7).

```
// Fig. 8.6: fig08_06.cpp
// Driver for class Date
#include <iostream.h>
#include "date1.h"

int main()
{
   Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 5 of 7).

```
   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3 << "\n\n";

   cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";

   d3.setDate( 2, 28, 1992 );
   cout << "  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << "\n\n";

   Date d4( 3, 18, 1969 );

   cout << "Testing the preincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "++d4 is " << ++d4 << '\n';
   cout << "  d4 is " << d4 << "\n\n";

   cout << "Testing the postincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "d4++ is " << d4++ << '\n';
   cout << "  d4 is " << d4 << endl;

   return 0;
}
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 6 of 7).

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Fig. 8.6 Output from driver for class **`Date`** (part 7 of 7).

```cpp
// Fig. 8.7: complex1.h
// Definition of class Complex
#ifndef COMPLEX1_H
#define COMPLEX1_H

class Complex {
public:
   Complex( double = 0.0, double = 0.0 );       // constructor
   Complex operator+( const Complex & ) const;  // addition
   Complex operator-( const Complex & ) const;  // subtraction
   const Complex &operator=( const Complex & ); // assignment
   void print() const;                          // output
private:
   double real;       // real part
   double imaginary;  // imaginary part
};

#endif
```

Fig. 8.7 Demonstrating class **`Complex`** (part 1 of 5).

```cpp
// Fig. 8.7: complex1.cpp
// Member function definitions for class Complex
#include <iostream.h>
#include "complex1.h"

// Constructor
Complex::Complex( double r, double i )
   : real( r ), imaginary( i ) { }

// Overloaded addition operator
Complex Complex::operator+( const Complex &operand2 ) const
{
   return Complex( real + operand2.real,
                   imaginary + operand2.imaginary );
}

```

```
// Overloaded subtraction operator
Complex Complex::operator-( const Complex &operand2 ) const
{
   return Complex( real - operand2.real,
                   imaginary - operand2.imaginary );
}
```

Fig. 8.7 Demonstrating class **`Complex`** (part 2 of 5).

```

// Overloaded = operator
const Complex& Complex::operator=( const Complex &right )
{
   real = right.real;
   imaginary = right.imaginary;
   return *this;   // enables cascading
}

// Display a Complex object in the form: (a, b)
void Complex::print() const
   { cout << '(' << real << ", " << imaginary << ')'; }
```

Fig. 8.7 Demonstrating class **`Complex`** (part 3 of 5).

```
// Fig. 8.7: fig08_07.cpp
// Driver for class Complex
#include <iostream.h>
#include "complex1.h"

int main()
{
   Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );

   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();

   x = y + z;
   cout << "\n\nx = y + z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();

   x = y - z;
   cout << "\n\nx = y - z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << endl;

   return 0;
}
```

Fig. 8.7 Demonstrating class **`Complex`** (part 4 of 5).

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Fig. 8.7 Demonstrating class **Complex** (part 5 of 5).

```
// Fig. 8.8: hugeint1.h
// Definition of the HugeInt class
#ifndef HUGEINT1_H
#define HUGEINT1_H

#include <iostream.h>

class HugeInt {
   friend ostream &operator<<( ostream &, HugeInt & );
public:
   HugeInt( long = 0 );       // conversion/default constructor
   HugeInt( const char * );            // conversion constructor
   HugeInt operator+( HugeInt & );     // add another HugeInt
   HugeInt operator+( int );           // add an int
   HugeInt operator+( const char * ); // add an int in a char *
private:
   short integer[30];
};

#endif
```

Fig. 8.8 A user-defined huge integer class (part 1 of 5).

```
// Fig. 8.8: hugeint1.cpp
// Member and friend function definitions for class HugeInt
#include <string.h>
#include "hugeint1.h"

// Conversion constructor
HugeInt::HugeInt( long val )
{
   int i;

   for ( i = 0; i <= 29; i++ )
      integer[ i ] = 0;   // initialize array to zero

   for ( i = 29; val != 0 && i >= 0; i-- ) {
      integer[ i ] = val % 10;
      val /= 10;
   }
}

HugeInt::HugeInt( const char *string )
{
   int i, j;

```

```
   for ( i = 0; i <= 29; i++ )
      integer[ i ] = 0;

   for ( i = 30 - strlen( string ), j = 0; i <= 29; i++, j++ )
      integer[ i ] = string[ j ] - '0';
}

// Addition
HugeInt HugeInt::operator+( HugeInt &op2 )
{
   HugeInt temp;
   int carry = 0;

   for ( int i = 29; i >= 0; i-- ) {
      temp.integer[ i ] = integer[ i ] +
                          op2.integer[ i ] + carry;

      if ( temp.integer[ i ] > 9 ) {
         temp.integer[ i ] %= 10;
         carry = 1;
      }
      else
         carry = 0;
   }

   return temp;
}
```

Fig. 8.8 A user-defined huge integer class (part 2 of 5).

```

// Addition
HugeInt HugeInt::operator+( int op2 )
   { return *this + HugeInt( op2 ); }

// Addition
HugeInt HugeInt::operator+( const char *op2 )
   { return *this + HugeInt( op2 ); }

ostream& operator<<( ostream &output, HugeInt &num )
{
   int i;

   for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
      ; // skip leading zeros

   if ( i == 30 )
      output << 0;
   else
      for ( ; i <= 29; i++ )
         output << num.integer[ i ];

   return output;
}
```

Fig. 8.8 A user-defined huge integer class (part 3 of 5).

```
// Fig. 8.8: fig08_08.cpp
// Test driver for HugeInt class
#include <iostream.h>
#include "hugeint1.h"

int main()
{
   HugeInt n1( 7654321 ), n2( 7891234 ),
           n3( "99999999999999999999999999999" ),
           n4( "1" ), n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
        << "\nn3 is " << n3 << "\nn4 is " << n4
        << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";

   cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 )
        << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
```

Fig. 8.8 A user-defined huge integer class (part 4 of 5).

```

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << endl;

   return 0;
}
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Fig. 8.8 A user-defined huge integer class (part 5 of 5).

Illustrations List *(Main Page)*

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Fig. 9.1 Some simple inheritance examples.

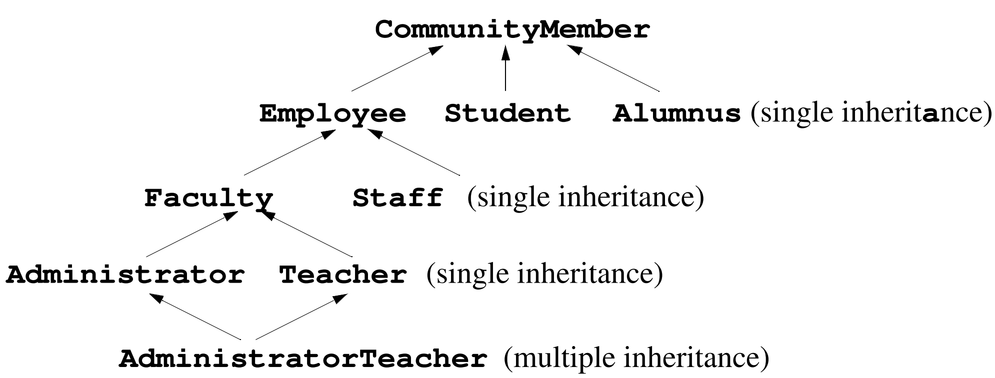


Fig. 9.2 An inheritance hierarchy for university community members.

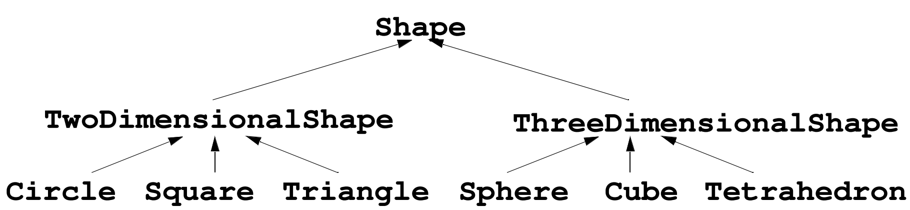


Fig. 9.3 A portion of a **`Shape`** class hierarchy.

```
// Fig. 9.4: point.h
// Definition of class Point
#ifndef POINT_H
#define POINT_H

class Point {
   friend ostream &operator<<( ostream &, const Point & );
public:
   Point( int = 0, int = 0 );      // default constructor
   void setPoint( int, int );      // set coordinates
   int getX() const { return x; }  // get x coordinate
   int getY() const { return y; }  // get y coordinate
protected:         // accessible by derived classes
   int x, y;       // x and y coordinates of the Point
};

#endif
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 1 of 6).

```
// Fig. 9.4: point.cpp
// Member functions for class Point
#include <iostream.h>
#include "point.h"

// Constructor for class Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Set x and y coordinates of Point
void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

// Output Point (with overloaded stream insertion operator)
ostream &operator<<( ostream &output, const Point &p )
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;   // enables cascaded calls
}
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 2 of 6).

```
// Fig. 9.4: circle.h
// Definition of class Circle
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream.h>
#include <iomanip.h>
#include "point.h"

class Circle : public Point {  // Circle inherits from Point
   friend ostream &operator<<( ostream &, const Circle & );
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius
   double area() const;        // calculate area
```

```
protected:
   double radius;
};

#endif
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 3 of 6).

```
// Fig. 9.4: circle.cpp
// Member function definitions for class Circle
#include "circle.h"

// Constructor for Circle calls constructor for Point
// with a member initializer then initializes radius.
Circle::Circle( double r, int a, int b )
   : Point( a, b )       // call base-class constructor
{ setRadius( r ); }

// Set radius of Circle
void Circle::setRadius( double r )
   { radius = ( r >= 0 ? r : 0 ); }

// Get radius of Circle
double Circle::getRadius() const { return radius; }

// Calculate area of Circle
double Circle::area() const
   { return 3.14159 * radius * radius; }

// Output a Circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<( ostream &output, const Circle &c )
{
   output << "Center = " << static_cast< Point >( c )
          << "; Radius = "
          << setiosflags( ios::fixed | ios::showpoint )
          << setprecision( 2 ) << c.radius;

   return output;   // enables cascaded calls
}
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 4 of 6).

```
// Fig. 9.4: fig09_04.cpp
// Casting base-class pointers to derived-class pointers
#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"

int main()
{
   Point *pointPtr = 0, p( 30, 50 );
   Circle *circlePtr = 0, c( 2.7, 120, 89 );

   cout << "Point p: " << p << "\nCircle c: " << c << '\n';

   // Treat a Circle as a Point (see only the base class part)
   pointPtr = &c;   // assign address of Circle to pointPtr
   cout << "\nCircle c (via *pointPtr): "
        << *pointPtr << '\n';

   // Treat a Circle as a Circle (with some casting)
```

```
   pointPtr = &c;   // assign address of Circle to pointPtr

   // cast base-class pointer to derived-class pointer
   circlePtr = static_cast< Circle * >( pointPtr );
   cout << "\nCircle c (via *circlePtr):\n" << *circlePtr
        << "\nArea of c (via circlePtr): "
        << circlePtr->area() << '\n';

   // DANGEROUS: Treat a Point as a Circle
   pointPtr = &p;   // assign address of Point to pointPtr

   // cast base-class pointer to derived-class pointer
   circlePtr = static_cast< Circle * >( pointPtr );
   cout << "\nPoint p (via *circlePtr):\n" << *circlePtr
        << "\nArea of object circlePtr points to: "
        << circlePtr->area() << endl;
   return 0;
}
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 5 of 6).

```
Point p: [30, 50]
Circle c: Center = [120, 89]; Radius = 2.70

Circle c (via *pointPtr): [120, 89]

Circle c (via *circlePtr):
Center = [120, 89]; Radius = 2.70
Area of c (via circlePtr): 22.90

Point p (via *circlePtr):
Center = [30, 50]; Radius = 0.00
Area of object circlePtr points to: 0.00
```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 6 of 6).

```
// Fig. 9.5: employ.h
// Definition of class Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee( const char *, const char * );  // constructor
   void print() const;  // output first and last name
   ~Employee();         // destructor
private:
   char *firstName;     // dynamically allocated string
   char *lastName;      // dynamically allocated string
};

#endif
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 1 of 5).

```
// Fig. 9.5: employ.cpp
// Member function definitions for class Employee
#include <string.h>
#include <iostream.h>
#include <assert.h>
#include "employ.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 ); // terminate if not allocated
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );  // terminate if not allocated
   strcpy( lastName, last );
}

// Output employee name
void Employee::print() const
   { cout << firstName << ' ' << lastName; }

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;    // reclaim dynamic memory
   delete [] lastName;     // reclaim dynamic memory
}
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 2 of 5).

```
// Fig. 9.5: hourly.h
// Definition of class HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker( const char*, const char*, double, double );
   double getPay() const;  // calculate and return salary
   void print() const;     // overridden base-class print
private:
   double wage;            // wage per hour
   double hours;           // hours worked for week
};

#endif
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 3 of 5).

```
// Fig. 9.5: hourly.cpp
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include <iomanip.h>
#include "hourly.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker( const char *first,
                            const char *last,
                            double initHours, double initWage )
   : Employee( first, last )   // call base-class constructor
{
   hours = initHours;  // should validate
   wage = initWage;    // should validate
}

// Get the HourlyWorker's pay
double HourlyWorker::getPay() const { return wage * hours; }

// Print the HourlyWorker's name and pay
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print() is executing\n\n";
   Employee::print();   // call base-class print function

   cout << " is an hourly worker with pay of $"
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 ) << getPay() << endl;
}
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 4 of 5).

```
// Fig. 9.5: fig.09_05.cpp
// Overriding a base-class member function in a
// derived class.
#include <iostream.h>
#include "hourly.h"

int main()
{
   HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
   h.print();
   return 0;
}
```

```
HourlyWorker::print() is executing

Bob Smith is an hourly worker with pay of $400.00
```

Fig. 9.5 Overriding a base-class member function in a derived class (part 5 of 5).

Base class member access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
public	**public** in derived class. Can be accessed directly by any non-**static** member functions, **friend** functions and non-member functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
protected	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
private	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.

Fig. 9.6 Summary of base-class member accessibility in a derived class.

```
// Fig. 9.7: point2.h
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
public:
   Point( int = 0, int = 0 );  // default constructor
   ~Point();     // destructor
protected:       // accessible by derived classes
   int x, y;     // x and y coordinates of Point
};

#endif
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 1 of 5).

```
// Fig. 9.7: point2.cpp
// Member function definitions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point( int a, int b )
{
   x = a;
   y = b;

   cout << "Point  constructor: "
        << '[' << x << ", " << y << ']' << endl;
}

// Destructor for class Point
Point::~Point()
{
   cout << "Point  destructor:  "
        << '[' << x << ", " << y << ']' << endl;
}
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 2 of 5).

```
// Fig. 9.7: circle2.h
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );

   ~Circle();
private:
   double radius;
};

#endif
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 3 of 5).

```
// Fig. 9.7: circle2.cpp
// Member function definitions for class Circle
#include "circle2.h"

// Constructor for Circle calls constructor for Point
Circle::Circle( double r, int a, int b )
   : Point( a, b )   // call base-class constructor
{
   radius = r;  // should validate
   cout << "Circle constructor: radius is "
        << radius << " [" << x << ", " << y << ']' << endl;
}

// Destructor for class Circle
Circle::~Circle()
{
   cout << "Circle destructor:  radius is "
        << radius << " [" << x << ", " << y << ']' << endl;
}
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 4 of 5).

```
// Fig. 9.7: fig09_07.cpp
// Demonstrate when base-class and derived-class
// constructors and destructors are called.
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

int main()
{
   // Show constructor and destructor calls for Point
   {
      Point p( 11, 22 );
   }

   cout << endl;
   Circle circle1( 4.5, 72, 29 );
   cout << endl;
   Circle circle2( 10, 5, 5 );
   cout << endl;
   return 0;
}
```

```
Point  constructor: [11, 22]
Point  destructor:  [11, 22]

Point  constructor: [72, 29]
Circle constructor: radius is 4.5 [72, 29]

Point  constructor: [5, 5]
Circle constructor: radius is 10 [5, 5]

Circle destructor:  radius is 10 [5, 5]
Point  destructor:  [5, 5]
Circle destructor:  radius is 4.5 [72, 29]
Point  destructor:  [72, 29]
```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 5 of 5).

```
// Fig. 9.8: point2.h
// Definition of class Point
#ifndef POINT2_H
#define POINT2_H

class Point {
   friend ostream &operator<<( ostream &, const Point & );
public:
   Point( int = 0, int = 0 );      // default constructor
   void setPoint( int, int );      // set coordinates
   int getX() const { return x; }  // get x coordinate
   int getY() const { return y; }  // get y coordinate
protected:        // accessible to derived classes
   int x, y;      // coordinates of the point
};

#endif
```

Fig. 9.8 Demonstrating class **Point** (part 1 of 3).

```
// Fig. 9.8: point2.cpp
// Member functions for class Point
#include <iostream.h>
#include "point2.h"

// Constructor for class Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Set the x and y coordinates
void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

// Output the Point
ostream &operator<<( ostream &output, const Point &p )
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;          // enables cascading
}
```

Fig. 9.8 Demonstrating class **Point** (part 2 of 3).

```
// Fig. 9.8: fig09_08.cpp
// Driver for class Point
#include <iostream.h>
#include "point2.h"

int main()
{
   Point p( 72, 115 );   // instantiate Point object p

   // protected data of Point inaccessible to main
   cout << "X coordinate is " << p.getX()
        << "\nY coordinate is " << p.getY();

   p.setPoint( 10, 10 );
   cout << "\n\nThe new location of p is " << p << endl;

   return 0;
}
```

```
X coordinate is 72
Y coordinate is 115

The new location of p is [10, 10]
```

Fig. 9.8 Demonstrating class **Point** (part 3 of 3).

```
// Fig. 9.9: circle2.h
// Definition of class Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<( ostream &, const Circle & );
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );
   void setRadius( double );    // set radius
   double getRadius() const;    // return radius
   double area() const;         // calculate area
protected:          // accessible to derived classes
   double radius;   // radius of the Circle
};

#endif
```

Fig. 9.9 Demonstrating class **Circle** (part 1 of 5).

```
// Fig. 9.9: circle2.cpp
// Member function definitions for class Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Constructor for Circle calls constructor for Point
// with a member initializer and initializes radius
Circle::Circle( double r, int a, int b )
   : Point( a, b )       // call base-class constructor
{ setRadius( r ); }

// Set radius
void Circle::setRadius( double r )
   { radius = ( r >= 0 ? r : 0 ); }

// Get radius
double Circle::getRadius() const { return radius; }
```

Fig. 9.9 Demonstrating class **Circle** (part 2 of 5).

```
// Calculate area of Circle
double Circle::area() const
   { return 3.14159 * radius * radius; }

// Output a circle in the form:
// Center = [x, y]; Radius = #.##
ostream &operator<<( ostream &output, const Circle &c )
{
   output << "Center = " << static_cast< Point > ( c )
          << "; Radius = "
          << setiosflags( ios::fixed | ios::showpoint )
          << setprecision( 2 ) << c.radius;

   return output;   //  enables cascaded calls
}
```

Fig. 9.9 Demonstrating class **Circle** (part 3 of 5).

```
// Fig. 9.9: fig09_09.cpp
// Driver for class Circle
#include <iostream.h>
#include "point2.h"
#include "circle2.h"

int main()
{
   Circle c( 2.5, 37, 43 );

   cout << "X coordinate is " << c.getX()
        << "\nY coordinate is " << c.getY()
        << "\nRadius is " << c.getRadius();

   c.setRadius( 4.25 );
   c.setPoint( 2, 2 );
   cout << "\n\nThe new location and radius of c are\n"
        << c << "\nArea " << c.area() << '\n';

   Point &pRef = c;
   cout << "\nCircle printed as a Point is: " << pRef << endl;

   return 0;
}
```

Fig. 9.9 Demonstrating class **Circle** (part 4 of 5).

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area 56.74

Circle printed as a Point is: [2, 2]
```

Fig. 9.9 Demonstrating class **Circle** (part 5 of 5).

```
// Fig. 9.10: cylindr2.h
// Definition of class Cylinder
#ifndef CYLINDR2_H
#define CYLINDR2_H

#include "circle2.h"

class Cylinder : public Circle {
   friend ostream &operator<<( ostream &, const Cylinder & );

public:
   // default constructor
   Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

   void setHeight( double );    // set height
   double getHeight() const;    // return height
   double area() const;         // calculate and return area
   double volume() const;       // calculate and return volume

protected:
   double height;               // height of the Cylinder
};

#endif
```

Fig. 9.10 Demonstrating class **Cylinder** (part 1 of 5).

```
// Fig. 9.10: cylindr2.cpp
// Member and friend function definitions
// for class Cylinder.
#include <iostream.h>
#include <iomanip.h>
#include "cylindr2.h"

```

Fig. 9.10 Demonstrating class **Cylinder** (part 2 of 5).

```
// Cylinder constructor calls Circle constructor
Cylinder::Cylinder( double h, double r, int x, int y )
   : Circle( r, x, y )   // call base-class constructor
{ setHeight( h ); }

// Set height of Cylinder
void Cylinder::setHeight( double h )
   { height = ( h >= 0 ? h : 0 ); }

// Get height of Cylinder
double Cylinder::getHeight() const { return height; }

// Calculate area of Cylinder (i.e., surface area)
double Cylinder::area() const
{
   return 2 * Circle::area() +
          2 * 3.14159 * radius * height;
}

// Calculate volume of Cylinder
double Cylinder::volume() const
   { return Circle::area() * height; }

// Output Cylinder dimensions
ostream &operator<<( ostream &output, const Cylinder &c )
```

```
{
   output << static_cast< Circle >( c )
          << "; Height = " << c.height;

   return output;   // enables cascaded calls
}
```

Fig. 9.10 Demonstrating class **Cylinder** (part 3 of 5).

```
// Fig. 9.10: fig09_10.cpp
// Driver for class Cylinder
#include <iostream.h>
#include <iomanip.h>
#include "point2.h"
#include "circle2.h"
#include "cylindr2.h"

int main()
{
   // create Cylinder object
   Cylinder cyl( 5.7, 2.5, 12, 23 );

```

Fig. 9.10 Demonstrating class **Cylinder** (part 4 of 5).

```
   // use get functions to display the Cylinder
   cout << "X coordinate is " << cyl.getX()
        << "\nY coordinate is " << cyl.getY()
        << "\nRadius is " << cyl.getRadius()
        << "\nHeight is " << cyl.getHeight() << "\n\n";

   // use set functions to change the Cylinder's attributes
   cyl.setHeight( 10 );
   cyl.setRadius( 4.25 );
   cyl.setPoint( 2, 2 );
   cout << "The new location, radius, and height of cyl are:\n"
        << cyl << '\n';

   // display the Cylinder as a Point
   Point &pRef = cyl;    // pRef "thinks" it is a Point
   cout << "\nCylinder printed as a Point is: "
        << pRef << "\n\n";

   // display the Cylinder as a Circle
   Circle &circleRef = cyl;  // circleRef thinks it is a Circle
   cout << "Cylinder printed as a Circle is:\n" << circleRef
        << "\nArea: " << circleRef.area() << endl;

   return 0;
}
```

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00

Cylinder printed as a Point is: [2, 2]

Cylinder printed as a Circle is:
Center = [2, 2]; Radius = 4.25
Area: 56.74
```

Fig. 9.10 Demonstrating class **Cylinder** (part 5 of 5).

```
// Fig. 9.11: base1.h
// Definition of class Base1
#ifndef BASE1_H
#define BASE1_H

class Base1 {
public:
   Base1( int x ) { value = x; }
   int getData() const { return value; }
protected:      // accessible to derived classes
   int value;   // inherited by derived class
};

#endif
```

Fig. 9.11 Demonstrating multiple inheritance (part 1 of 6).

```
// Fig. 9.11: base2.h
// Definition of class Base2
#ifndef BASE2_H
#define BASE2_H

class Base2 {
public:
   Base2( char c ) { letter = c; }
   char getData() const { return letter; }
protected:        // accessible to derived classes
   char letter;   // inherited by derived class
};

#endif
```

Fig. 9.11 Demonstrating multiple inheritance (part 2 of 6).

```
// Fig. 9.11: derived.h
// Definition of class Derived which inherits
// multiple base classes (Base1 and Base2).
#ifndef DERIVED_H
#define DERIVED_H

#include "base1.h"
#include "base2.h"

// multiple inheritance
class Derived : public Base1, public Base2 {
   friend ostream &operator<<( ostream &, const Derived & );

public:
   Derived( int, char, double );
   double getReal() const;

private:
   double real;   // derived class's private data
};

#endif
```

Fig. 9.11 Demonstrating multiple inheritance (part 3 of 6).

```
// Fig. 9.11: derived.cpp
// Member function definitions for class Derived
#include <iostream.h>
#include "derived.h"

// Constructor for Derived calls constructors for
// class Base1 and class Base2.
// Use member initializers to call base-class constructors
Derived::Derived( int i, char c, double f )
   : Base1( i ), Base2( c ), real ( f ) { }

// Return the value of real
double Derived::getReal() const { return real; }

// Display all the data members of Derived
ostream &operator<<( ostream &output, const Derived &d )
{
   output << "    Integer: " << d.value
          << "\n  Character: " << d.letter
          << "\nReal number: " << d.real;

   return output;   // enables cascaded calls
}
```

Fig. 9.11 Demonstrating multiple inheritance (part 4 of 6).

```
// Fig. 9.11: fig09_11.cpp
// Driver for multiple inheritance example
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

int main()
{
   Base1 b1( 10 ), *base1Ptr = 0;  // create Base1 object
   Base2 b2( 'Z' ), *base2Ptr = 0; // create Base2 object
   Derived d( 7, 'A', 3.5 );       // create Derived object

   // print data members of base class objects
   cout << "Object b1 contains integer " << b1.getData()
        << "\nObject b2 contains character " << b2.getData()
        << "\nObject d contains:\n" << d << "\n\n";

   // print data members of derived class object
   // scope resolution operator resolves getData ambiguity
   cout << "Data members of Derived can be"
        << " accessed individually:"
        << "\n    Integer: " << d.Base1::getData()
        << "\n  Character: " << d.Base2::getData()
        << "\nReal number: " << d.getReal() << "\n\n";

   cout << "Derived can be treated as an "
        << "object of either base class:\n";

   // treat Derived as a Base1 object
   base1Ptr = &d;
   cout << "base1Ptr->getData() yields "
        << base1Ptr->getData() << '\n';

   // treat Derived as a Base2 object
   base2Ptr = &d;
   cout << "base2Ptr->getData() yields "
        << base2Ptr->getData() << endl;

   return 0;
}
```

Fig. 9.11 Demonstrating multiple inheritance (part 5 of 6).

```
Object b1 contains integer 10
Object b2 contains character Z
Object d contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 9.11 Demonstrating multiple inheritance (part 6 of 6).

Illustrations List *(Main Page)*

```
// Fig. 10.1: employ2.h
// Abstract base class Employee
#ifndef EMPLOY2_H
#define EMPLOY2_H

#include <iostream.h>

class Employee {
public:
   Employee( const char *, const char * );
   ~Employee();    // destructor reclaims memory
   const char *getFirstName() const;
   const char *getLastName() const;

   // Pure virtual function makes Employee abstract base class
   virtual double earnings() const = 0;   // pure virtual
   virtual void print() const;            // virtual
private:
   char *firstName;
   char *lastName;
};

#endif
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 1 of 13).

```
// Fig. 10.1: employ2.cpp
// Member function definitions for
// abstract base class Employee.
// Note: No definitions given for pure virtual functions.
#include <string.h>
#include <assert.h>
#include "employ2.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 );    // test that new worked
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );     // test that new worked
   strcpy( lastName, last );
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}

// Return a pointer to the first name
// Const return type prevents caller from modifying private
// data. Caller should copy returned string before destructor
// deletes dynamic storage to prevent undefined pointer.
const char *Employee::getFirstName() const
{
   return firstName;   // caller must delete memory
}
```

```

// Return a pointer to the last name
// Const return type prevents caller from modifying private
// data. Caller should copy returned string before destructor
// deletes dynamic storage to prevent undefined pointer.
const char *Employee::getLastName() const
{
   return lastName;   // caller must delete memory
}

// Print the name of the Employee
void Employee::print() const
   { cout << firstName << ' ' << lastName; }
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 2 of 13).

```
// Fig. 10.1: boss1.h
// Boss class derived from Employee
#ifndef BOSS1_H
#define BOSS1_H
#include "employ2.h"

class Boss : public Employee {
public:
   Boss( const char *, const char *, double = 0.0 );
   void setWeeklySalary( double );
   virtual double earnings() const;
   virtual void print() const;
private:
   double weeklySalary;
};

#endif
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 3 of 13).

```
// Fig. 10.1: boss1.cpp
// Member function definitions for class Boss
#include "boss1.h"

// Constructor function for class Boss
Boss::Boss( const char *first, const char *last, double s )
   : Employee( first, last )  // call base-class constructor
{ setWeeklySalary( s ); }

// Set the Boss's salary
void Boss::setWeeklySalary( double s )
   { weeklySalary = s > 0 ? s : 0; }

// Get the Boss's pay
double Boss::earnings() const { return weeklySalary; }

// Print the Boss's name
void Boss::print() const
{
   cout << "\n             Boss: ";
   Employee::print();
}
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 4 of 13).

```
// Fig. 10.1: commis1.h
// CommissionWorker class derived from Employee
#ifndef COMMIS1_H
#define COMMIS1_H
#include "employ2.h"

class CommissionWorker : public Employee {
public:
   CommissionWorker( const char *, const char *,
                     double = 0.0, double = 0.0,
                     int = 0 );
   void setSalary( double );
   void setCommission( double );
   void setQuantity( int );
   virtual double earnings() const;
   virtual void print() const;
private:
   double salary;       // base salary per week
   double commission;   // amount per item sold
   int quantity;        // total items sold for week
};

#endif
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 5 of 13).

```
// Fig. 10.1: commis1.cpp
// Member function definitions for class CommissionWorker
#include <iostream.h>
#include "commis1.h"

// Constructor for class CommissionWorker
CommissionWorker::CommissionWorker( const char *first,
        const char *last, double s, double c, int q )
   : Employee( first, last )  // call base-class constructor
{
   setSalary( s );
   setCommission( c );
   setQuantity( q );
}

// Set CommissionWorker's weekly base salary
void CommissionWorker::setSalary( double s )
   { salary = s > 0 ? s : 0; }

// Set CommissionWorker's commission
void CommissionWorker::setCommission( double c )
   { commission = c > 0 ? c : 0; }

// Set CommissionWorker's quantity sold
void CommissionWorker::setQuantity( int q )
   { quantity = q > 0 ? q : 0; }

// Determine CommissionWorker's earnings
double CommissionWorker::earnings() const
   { return salary + commission * quantity; }

// Print the CommissionWorker's name
void CommissionWorker::print() const
{
   cout << "\nCommission worker: ";
   Employee::print();
```

```
}
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 6 of 13).

```
// Fig. 10.1: piece1.h
// PieceWorker class derived from Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
   PieceWorker( const char *, const char *,
                double = 0.0, int = 0);
   void setWage( double );
   void setQuantity( int );
   virtual double earnings() const;
   virtual void print() const;
private:
   double wagePerPiece; // wage for each piece output
   int quantity;        // output for week
};

#endif
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 7 of 13).

```
// Fig. 10.1: piece1.cpp
// Member function definitions for class PieceWorker
#include <iostream.h>
#include "piece1.h"

// Constructor for class PieceWorker
PieceWorker::PieceWorker( const char *first, const char *last,
                          double w, int q )
   : Employee( first, last )  // call base-class constructor
{
   setWage( w );
   setQuantity( q );
}

// Set the wage
void PieceWorker::setWage( double w )
   { wagePerPiece = w > 0 ? w : 0; }

// Set the number of items output
void PieceWorker::setQuantity( int q )
   { quantity = q > 0 ? q : 0; }

// Determine the PieceWorker's earnings
double PieceWorker::earnings() const
   { return quantity * wagePerPiece; }
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 8 of 13).

```

// Print the PieceWorker's name
void PieceWorker::print() const
{
   cout << "\n     Piece worker: ";
```

```
   Employee::print();
}
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 9 of 13).

```
// Fig. 10.1: hourly1.h
// Definition of class HourlyWorker
#ifndef HOURLY1_H
#define HOURLY1_H
#include "employ2.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker( const char *, const char *,
                 double = 0.0, double = 0.0);
   void setWage( double );
   void setHours( double );
   virtual double earnings() const;
   virtual void print() const;
private:
   double wage;    // wage per hour
   double hours;   // hours worked for week
};

#endif
```

Fig. 10.1 Demonstrating polymorphism with the **`Employee`** class hierarchy (part 10 of 13).

```
// Fig. 10.1: hourly1.cpp
// Member function definitions for class HourlyWorker
#include <iostream.h>
#include "hourly1.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker( const char *first,
                            const char *last,
                            double w, double h )
   : Employee( first, last )   // call base-class constructor
{
   setWage( w );
   setHours( h );
}

// Set the wage
void HourlyWorker::setWage( double w )
   { wage = w > 0 ? w : 0; }

// Set the hours worked
void HourlyWorker::setHours( double h )
   { hours = h >= 0 && h < 168 ? h : 0; }

// Get the HourlyWorker's pay
double HourlyWorker::earnings() const
{
   if ( hours <= 40 ) // no overtime
      return wage * hours;
   else               // overtime is paid at wage * 1.5
      return 40 * wage + ( hours - 40 ) * wage * 1.5;
}

// Print the HourlyWorker's name
```

```
void HourlyWorker::print() const
{
   cout << "\n    Hourly worker: ";
   Employee::print();
}
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 11 of 13).

```
// Fig. 10.1: fig10_01.cpp
// Driver for Employee hierarchy
#include <iostream.h>
#include <iomanip.h>
#include "employ2.h"
#include "boss1.h"
#include "commis1.h"
#include "piece1.h"
#include "hourly1.h"

void virtualViaPointer( const Employee * );
void virtualViaReference( const Employee & );

int main()
{
   // set output formatting
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

   Boss b( "John", "Smith", 800.00 );
   b.print();                              // static binding
   cout << " earned $" << b.earnings();    // static binding
   virtualViaPointer( &b );          // uses dynamic binding
   virtualViaReference( b );         // uses dynamic binding

   CommissionWorker c( "Sue", "Jones", 200.0, 3.0, 150 );
   c.print();                              // static binding
   cout << " earned $" << c.earnings();    // static binding
   virtualViaPointer( &c );          // uses dynamic binding
   virtualViaReference( c );         // uses dynamic binding

   PieceWorker p( "Bob", "Lewis", 2.5, 200 );
   p.print();                              // static binding
   cout << " earned $" << p.earnings();    // static binding
   virtualViaPointer( &p );          // uses dynamic binding
   virtualViaReference( p );         // uses dynamic binding

   HourlyWorker h( "Karen", "Price", 13.75, 40 );
   h.print();                              // static binding
   cout << " earned $" << h.earnings();    // static binding
   virtualViaPointer( &h );          // uses dynamic binding
   virtualViaReference( h );         // uses dynamic binding
   cout << endl;
   return 0;
}
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 12 of 13).

```
// Make virtual function calls off a base-class pointer
// using dynamic binding.
void virtualViaPointer( const Employee *baseClassPtr )
{
   baseClassPtr->print();
   cout << " earned $" << baseClassPtr->earnings();
}

// Make virtual function calls off a base-class reference
// using dynamic binding.
void virtualViaReference( const Employee &baseClassRef )
{
   baseClassRef.print();
   cout << " earned $" << baseClassRef.earnings();
}
```

```
             Boss: John Smith earned $800.00
             Boss: John Smith earned $800.00
             Boss: John Smith earned $800.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 13 of 13).

```
// Fig. 10.2: shape.h
// Definition of abstract base class Shape
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream.h>

class Shape {
public:
   virtual double area() const { return 0.0; }
   virtual double volume() const { return 0.0; }

   // pure virtual functions overridden in derived classes
   virtual void printShapeName() const = 0;
   virtual void print() const = 0;
};

#endif
```

Fig. 10.2 Definition of abstract base class **Shape** (part 1 of 10).

```
// Fig. 10.2: point1.h
// Definition of class Point
#ifndef POINT1_H
#define POINT1_H
#include "shape.h"

class Point : public Shape {
public:
```

```
   Point( int = 0, int = 0 );  // default constructor
   void setPoint( int, int );
   int getX() const { return x; }
   int getY() const { return y; }
   virtual void printShapeName() const { cout << "Point: "; }
   virtual void print() const;
private:
   int x, y;   // x and y coordinates of Point
};

#endif
```

Fig. 10.2 Definition of class **Point** (part 2 of 10).

```
// Fig. 10.2: point1.cpp
// Member function definitions for class Point
#include "point1.h"

Point::Point( int a, int b ) { setPoint( a, b ); }

void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

void Point::print() const
   { cout << '[' << x << ", " << y << ']'; }
```

Fig. 10.2 Member function definitions for class **Point** (part 3 of 10).

```
// Fig. 10.2: circle1.h
// Definition of class Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );

   void setRadius( double );
   double getRadius() const;
   virtual double area() const;
   virtual void printShapeName() const { cout << "Circle: "; }
   virtual void print() const;
private:
   double radius;   // radius of Circle
};

#endif
```

Fig. 10.2 Definition of class **Circle** (part 4 of 10).

```
// Fig. 10.2: circle1.cpp
// Member function definitions for class Circle
#include "circle1.h"

Circle::Circle( double r, int a, int b )
   : Point( a, b )  // call base-class constructor
{ setRadius( r ); }

```

```
void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }

double Circle::getRadius() const { return radius; }

double Circle::area() const
   { return 3.14159 * radius * radius; }

void Circle::print() const
{
   Point::print();
   cout << "; Radius = " << radius;
}
```

Fig. 10.2 Member function definitions for class **Circle** (part 5 of 10).

```
// Fig. 10.2: cylindr1.h
// Definition of class Cylinder
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
public:
   // default constructor
   Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

   void setHeight( double );
   double getHeight() const;
   virtual double area() const;
   virtual double volume() const;
   virtual void printShapeName() const {cout << "Cylinder: ";}
   virtual void print() const;
private:
   double height;   // height of Cylinder
};

#endif
```

Fig. 10.2 Definition of class **Cylinder** (part 6 of 10).

```
// Fig. 10.2: cylindr1.cpp
// Member and friend function definitions for class Cylinder
#include "cylindr1.h"

Cylinder::Cylinder( double h, double r, int x, int y )
   : Circle( r, x, y )  // call base-class constructor
{ setHeight( h ); }

void Cylinder::setHeight( double h )
   { height = h > 0 ? h : 0; }

double Cylinder::getHeight() const { return height; }

double Cylinder::area() const
{
   // surface area of Cylinder
   return 2 * Circle::area() +
          2 * 3.14159 * getRadius() * height;
}

double Cylinder::volume() const
   { return Circle::area() * height; }
```

```
void Cylinder::print() const
{
   Circle::print();
   cout << "; Height = " << height;
}
```

Fig. 10.2 Member function definitions for class **Cylinder** (part 7 of 10).

```
// Fig. 10.2: fig10_02.cpp
// Driver for shape, point, circle, cylinder hierarchy
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );

int main()
{
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

```

Fig. 10.2 Driver for point, circle, cylinder hierarchy (part 8 of 10).

```
   Point point( 7, 11 );                    // create a Point
   Circle circle( 3.5, 22, 8 );             // create a Circle
   Cylinder cylinder( 10, 3.3, 10, 10 );    // create a Cylinder

   point.printShapeName();    // static binding
   point.print();             // static binding
   cout << '\n';

   circle.printShapeName();   // static binding
   circle.print();            // static binding
   cout << '\n';

   cylinder.printShapeName(); // static binding
   cylinder.print();          // static binding
   cout << "\n\n";

   Shape *arrayOfShapes[ 3 ];  // array of base-class pointers

   // aim arrayOfShapes[0] at derived-class Point object
   arrayOfShapes[ 0 ] = &point;

   // aim arrayOfShapes[1] at derived-class Circle object
   arrayOfShapes[ 1 ] = &circle;

   // aim arrayOfShapes[2] at derived-class Cylinder object
   arrayOfShapes[ 2 ] = &cylinder;

   // Loop through arrayOfShapes and call virtualViaPointer
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class pointers\n";

   for ( int i = 0; i < 3; i++ )
      virtualViaPointer( arrayOfShapes[ i ] );

   // Loop through arrayOfShapes and call virtualViaReference
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class references\n";

   for ( int j = 0; j < 3; j++ )
      virtualViaReference( *arrayOfShapes[ j ] );

   return 0;
}
```

Fig. 10.2 Driver for point, circle, cylinder hierarchy (part 9 of 10).

```
// Make virtual function calls off a base-class pointer
// using dynamic binding.
void virtualViaPointer( const Shape *baseClassPtr )
{
   baseClassPtr->printShapeName();
   baseClassPtr->print();
   cout << "\nArea = " << baseClassPtr->area()
        << "\nVolume = " << baseClassPtr->volume() << "\n\n";
}

// Make virtual function calls off a base-class reference
// using dynamic binding.
void virtualViaReference( const Shape &baseClassRef )
{
   baseClassRef.printShapeName();
   baseClassRef.print();
   cout << "\nArea = " << baseClassRef.area()
        << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```

```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00

Virtual function calls made off base-class pointers
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

Fig. 10.2 Driver for point, circle, cylinder hierarchy (part 10 of 10).

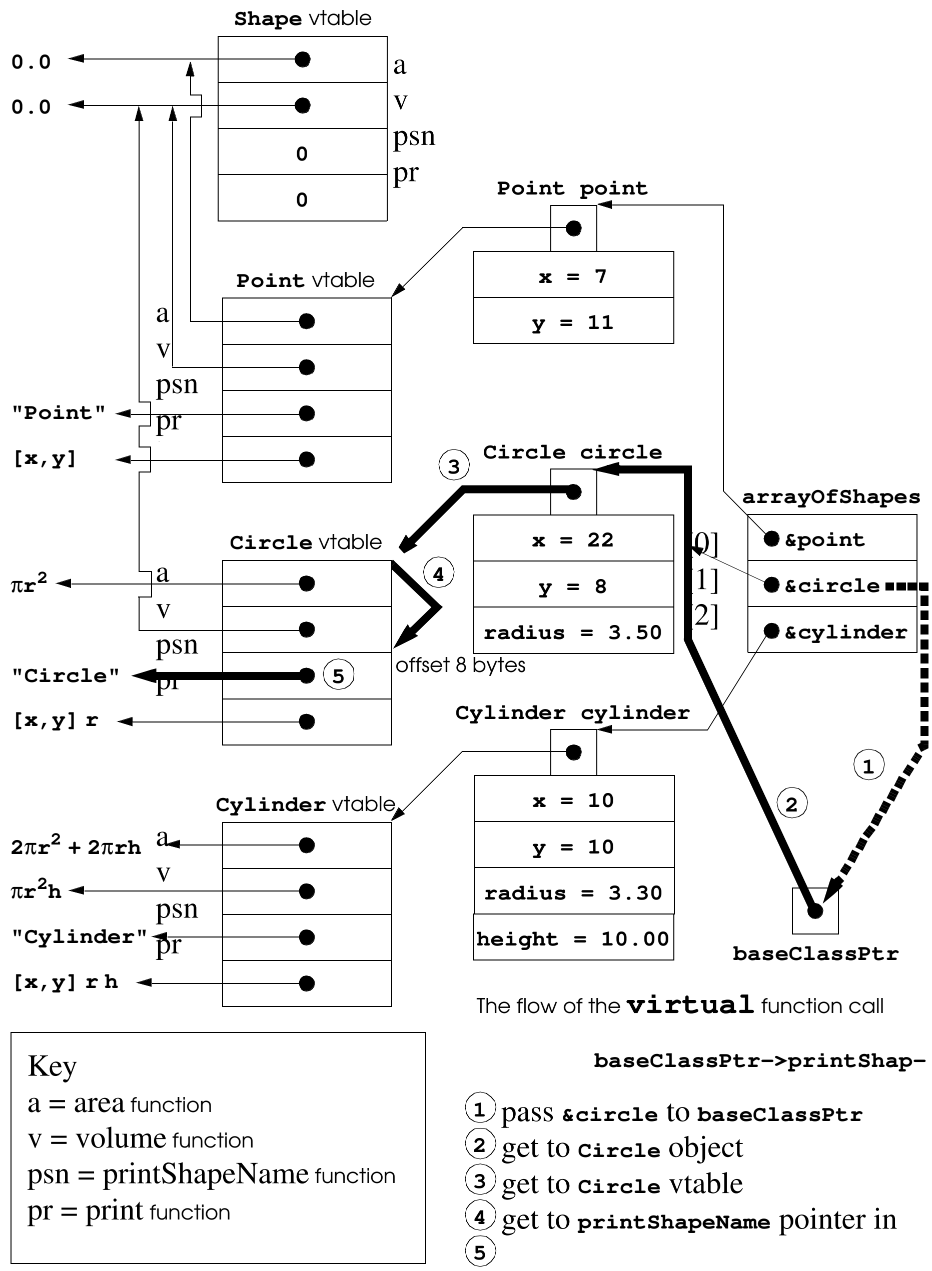


Fig. 10.3 Flow of control of a **virtual** function call.

Illustrations List ***(Main Page)***

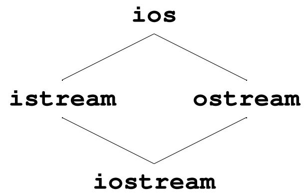


Fig. 11.1 Portion of the stream I/O class hierarchy.

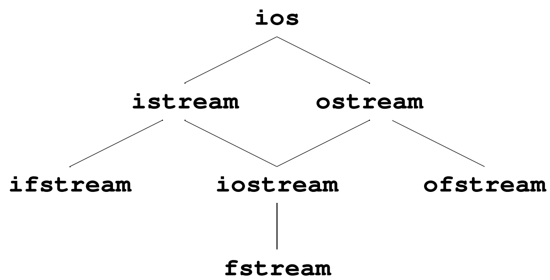


Fig. 11.2 Portion of stream-I/O class hierarchy with key file-processing classes.

```
// Fig. 11.3: fig11_03.cpp
// Outputting a string using stream insertion.
#include <iostream.h>

int main()
{
   cout << "Welcome to C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Fig. 11.3 Outputting a string using stream insertion.

```
// Fig. 11.4: fig11_04.cpp
// Outputting a string using two stream insertions.
#include <iostream.h>

int main()
{
   cout << "Welcome to ";
   cout << "C++!\n";

   return 0;
}
```

```
Welcome to C++!
```

Fig. 11.4 Outputting a string using two stream insertions.

```
// Fig. 11.5: fig11_05.cpp
// Using the endl stream manipulator.
#include <iostream.h>

int main()
{
   cout << "Welcome to ";
   cout << "C++!";
   cout << endl;   // end line stream manipulator

   return 0;
}
```

```
Welcome to C++!
```

Fig. 11.5 Using the **endl** stream manipulator.

```
// Fig. 11.6: fig11_06.cpp
// Outputting expression values.
#include <iostream.h>

int main()
{
   cout << "47 plus 53 is ";

   // parentheses not needed; used for clarity
   cout << ( 47 + 53 );      // expression
   cout << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Fig. 11.6 Outputting expression values.

```
// Fig. 11.7: fig11_07.cpp
// Cascading the overloaded << operator.
#include <iostream.h>

int main()
{
   cout << "47 plus 53 is " << ( 47 + 53 ) << endl;

   return 0;
}
```

```
47 plus 53 is 100
```

Fig. 11.7 Cascading the overloaded **<<** operator.

```
// Fig. 11.8: fig11_08.cpp
// Printing the address stored in a char* variable
#include <iostream.h>

int main()
{
   char *string = "test";

   cout << "Value of string is: " << string
        << "\nValue of static_cast< void *>( string ) is: "
        << static_cast< void *>( string ) << endl;
   return 0;
}
```

```
Value of string is: test
Value of static_cast< void *>( string ) is: 0x00416D50
```

Fig. 11.8 Printing the address stored in a **char *** variable.

```
// Fig. 11.9: fig11_09.cpp
// Calculating the sum of two integers input from the keyboard
// with the cin object and the stream-extraction operator.
#include <iostream.h>

int main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << "Sum of " << x << " and " << y << " is: "
        << ( x + y ) << endl;

   return 0;
}
```

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Fig. 11.9 Calculating the sum of two integers input from the keyboard with **cin** and the stream-extraction operator.

```
// Fig. 11.10: fig11_10.cpp
// Avoiding a precedence problem between the stream-insertion
// operator and the conditional operator.
// Need parentheses around the conditional expression.
#include <iostream.h>

int main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << x << ( x == y ? " is" : " is not" )
        << " equal to " << y << endl;

```

Fig. 11.10 Avoiding a precedence problem between the stream-insertion operator and the conditional operator (part 1 of 2).

```
   return 0;
}
```

```
Enter two integers: 7 5
7 is not equal to 5
```

```
Enter two integers: 8 8
8 is equal to 8
```

Fig. 11.10 Avoiding a precedence problem between the stream-insertion operator and the conditional operator (part 2 of 2).

```
// Fig. 11.11: fig11_11.cpp
// Stream-extraction operator returning false on end-of-file.
#include <iostream.h>

int main()
{
   int grade, highestGrade = -1;

   cout << "Enter grade (enter end-of-file to end): ";
   while ( cin >> grade ) {
      if ( grade > highestGrade )
         highestGrade = grade;

      cout << "Enter grade (enter end-of-file to end): ";
   }

   cout << "\n\nHighest grade is: " << highestGrade << endl;
   return 0;
}
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z
Highest grade is: 99
```

Fig. 11.11 Stream-extraction operator returning false on end-of-file.

```
// Fig. 11.12: fig11_12.cpp
// Using member functions get, put, and eof.
#include <iostream.h>

int main()
{
   char c;

   cout << "Before input, cin.eof() is " << cin.eof()
        << "\nEnter a sentence followed by end-of-file:\n";

   while ( ( c = cin.get() ) != EOF)
      cout.put( c );

   cout << "\nEOF in this system is: " << c;
   cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
   return 0;
}
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
EOF in this system is: -1
After input cin.eof() is 1
```

Fig. 11.12 Using member functions **get**, **put**, and **eof**.

```
// Fig. 11.13: fig11_13.cpp
// Contrasting input of a string with cin and cin.get.
#include <iostream.h>

int main()
{
   const int SIZE = 80;
   char buffer1[ SIZE ], buffer2[ SIZE ];

   cout << "Enter a sentence:\n";
   cin >> buffer1;
   cout << "\nThe string read with cin was:\n"
        << buffer1 << "\n\n";

   cin.get( buffer2, SIZE );
   cout << "The string read with cin.get was:\n"
        << buffer2 << endl;

   return 0;
}
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Fig. 11.13 Contrasting input of a string using **cin** with stream extraction and input with **cin.get**.

```
// Fig. 11.14: fig11_14.cpp
// Character input with member function getline.
#include <iostream.h>

int main()
{
   const SIZE = 80;
   char buffer[ SIZE ];

   cout << "Enter a sentence:\n";
   cin.getline( buffer, SIZE );

   cout << "\nThe sentence entered is:\n" << buffer << endl;
   return 0;
}
```

Fig. 11.14 Character input with member function **getline** (part 1 of 2).

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Fig. 11.14 Character input with member function **getline** (part 2 of 2).

```
// Fig. 11.15: fig11_15.cpp
// Unformatted I/O with read, gcount and write.
#include <iostream.h>

int main()
{
   const int SIZE = 80;
   char buffer[ SIZE ];

   cout << "Enter a sentence:\n";
   cin.read( buffer, 20 );
   cout << "\nThe sentence entered was:\n";
   cout.write( buffer, cin.gcount() );
   cout << endl;
   return 0;
}
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Fig. 11.15 Unformatted I/O with the **read**, **gcount** and **write** member functions.

```
// Fig. 11.16: fig11_16.cpp
// Using hex, oct, dec and setbase stream manipulators.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int n;

   cout << "Enter a decimal number: ";
   cin >> n;

   cout << n << " in hexadecimal is: "
        << hex << n << '\n'
        << dec << n << " in octal is: "
        << oct << n << '\n'
        << setbase( 10 ) << n << " in decimal is: "
        << n << endl;

   return 0;
}
```

Fig. 11.16 Using the **hex**, **oct**, **dec** and **setbase** stream manipulators (part 1 of 2).

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 11.16 Using the **hex**, **oct**, **dec** and **setbase** stream manipulators (part 2 of 2).

```cpp
// Fig. 11.17: fig11_17.cpp
// Controlling precision of floating-point values
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
   double root2 = sqrt( 2.0 );
   int places;

   cout << setiosflags( ios::fixed)
        << "Square root of 2 with precisions 0-9.\n"
        << "Precision set by the "
        << "precision member function:" << endl;

   for ( places = 0; places <= 9; places++ ) {
      cout.precision( places );
      cout << root2 << '\n';
   }

   cout << "\nPrecision set by the "
        << "setprecision manipulator:\n";

   for ( places = 0; places <= 9; places++ )
      cout << setprecision( places ) << root2 << '\n';

   return 0;
}
```

Fig. 11.17 Controlling precision of floating-point values (part 1 of 2).

```
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by the setprecision manipulator:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 11.17 Controlling precision of floating-point values (part 2 of 2).

```
// fig11_18.cpp
// Demonstrating the width member function
#include <iostream.h>

int main()
{
   int w = 4;
   char string[ 10 ];

   cout << "Enter a sentence:\n";
   cin.width( 5 );

   while ( cin >> string ) {
      cout.width( w++ );
      cout << string << endl;
      cin.width( 5 );
   }

   return 0;
}
```

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

Fig. 11.18 Demonstrating the **width** member function.

```
// Fig. 11.19: fig11_19.cpp
// Creating and testing user-defined, nonparameterized
// stream manipulators.
#include <iostream.h>

// bell manipulator (using escape sequence \a)
ostream& bell( ostream& output ) { return output << '\a'; }

// ret manipulator (using escape sequence \r)
ostream& ret( ostream& output ) { return output << '\r'; }

// tab manipulator (using escape sequence \t)
ostream& tab( ostream& output ) { return output << '\t'; }

// endLine manipulator (using escape sequence \n
// and the flush member function)
ostream& endLine( ostream& output )
{
   return output << '\n' << flush;
}

int main()
{
   cout << "Testing the tab manipulator:" << endLine
        << 'a' << tab << 'b' << tab << 'c' << endLine
        << "Testing the ret and bell manipulators:"
        << endLine << "..........";
   cout << bell;
   cout << ret << "-----" << endLine;
   return 0;
}
```

```
Testing the tab manipulator:
a       b       c
Testing the ret and bell manipulators:
-----.....
```

Fig. 11.19 Creating and testing user-defined, nonparameterized stream manipulators.

Format state flag	Description
`ios::skipws`	Skip whitespace characters on an input stream.
`ios::left`	Left justify output in a field. Padding characters appear to the right if necessary.
`ios::right`	Right justify output in a field. Padding characters appear to the left if necessary.
`ios::internal`	Indicate that a number's sign should be left justified in a field and a number's magnitude should be right justified in that same field (i.e., padding characters appear between the sign and the number).
`ios::dec`	Specify that integers should be treated as decimal (base 10) values.
`ios::oct`	Specify that integers should be treated as octal (base 8) values.
`ios::hex`	Specify that integers should be treated as hexadecimal (base 16) values.
`ios::showbase`	Specify that the base of a number to be output ahead of the number (a leading **0** for octals; a leading **0x** or **0X** for hexadecimals).
`ios::showpoint`	Specify that floating-point numbers should be output with a decimal point. This is normally used with **ios::fixed** to guarantee a certain number of digits to the right of the decimal point.
`ios::uppercase`	Specify that uppercase **X** should be used in the **0X** before a hexadecimal integer and that uppercase **E** should be used when representing a floating-point value in scientific notation.
`ios::showpos`	Specify that positive and negative numbers should be preceded by a **+** or **-** sign, respectively.
`ios::scientific`	Specify output of a floating-point value in scientific notation.
`ios::fixed`	Specify output of a floating-point value in fixed-point notation with a specific number of digits to the right of the decimal point.

Fig. 11.20 Format state flags.

```
// Fig. 11.21: fig11_21.cpp
// Controlling the printing of trailing zeros and decimal
// points for floating-point values.
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
   cout << "Before setting the ios::showpoint flag\n"
        << "9.9900 prints as: " << 9.9900
        << "\n9.9000 prints as: " << 9.9000
        << "\n9.0000 prints as: " << 9.0000
        << "\n\nAfter setting the ios::showpoint flag\n";
   cout.setf( ios::showpoint );
   cout << "9.9900 prints as: " << 9.9900
        << "\n9.9000 prints as: " << 9.9000
        << "\n9.0000 prints as: " << 9.0000 << endl;
   return 0;
}
```

```
Before setting the ios::showpoint flag
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After setting the ios::showpoint flag
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Fig. 11.21 Controlling the printing of trailing zeros and decimal points with float values.

```
// Fig. 11.22: fig11_22.cpp
// Left-justification and right-justification.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int x = 12345;

   cout << "Default is right justified:\n"
        << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
        << "\nUse setf to set ios::left:\n" << setw(10);

   cout.setf( ios::left, ios::adjustfield );
   cout << x << "\nUse unsetf to restore default:\n";
   cout.unsetf( ios::left );
   cout << setw( 10 ) << x
        << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
        << "\nUse setiosflags to set ios::left:\n"
        << setw( 10 ) << setiosflags( ios::left ) << x
        << "\nUse resetiosflags to restore default:\n"
        << setw( 10 ) << resetiosflags( ios::left )
        << x << endl;
   return 0;
}
```

```
Default is right justified:
     12345

USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
Use unsetf to restore default:
     12345

USING PARAMETERIZED STREAM MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
     12345
```

Fig. 11.22 Left-justification and right-justification.

```
// Fig. 11.23: fig11_23.cpp
// Printing an integer with internal spacing and
// forcing the plus sign.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   cout << setiosflags( ios::internal | ios::showpos )
        << setw( 10 ) << 123 << endl;
   return 0;
}
```

```
+      123
```

Fig. 11.23 Printing an integer with internal spacing and forcing the plus sign.

```
// Fig. 11.24: fig11_24.cpp
// Using the fill member function and the setfill
// manipulator to change the padding character for
// fields larger than the values being printed.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int x = 10000;

   cout << x << " printed as int right and left justified\n"
        << "and as hex with internal justification.\n"
        << "Using the default pad character (space):\n";
```

Fig. 11.24 Using the **`fill`** member function and the **`setfill`** manipulator to change the padding character for fields larger than the values being printed (part 1 of 2).

```
   cout.setf( ios::showbase );
   cout << setw( 10 ) << x << '\n';
   cout.setf( ios::left, ios::adjustfield );
   cout << setw( 10 ) << x << '\n';
   cout.setf( ios::internal, ios::adjustfield );
   cout << setw( 10 ) << hex << x;

   cout << "\n\nUsing various padding characters:\n";
   cout.setf( ios::right, ios::adjustfield );
   cout.fill( '*' );
   cout << setw( 10 ) << dec << x << '\n';
   cout.setf( ios::left, ios::adjustfield );
   cout << setw( 10 ) << setfill( '%' ) << x << '\n';
   cout.setf( ios::internal, ios::adjustfield );
   cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
   return 0;
}
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Fig. 11.24 Using the **`fill`** member function and the **`setfill`** manipulator to change the padding character for fields larger than the values being printed (part 2 of 2).

```
// Fig. 11.25: fig11_25.cpp
// Using the ios::showbase flag
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int x = 100;

   cout << setiosflags( ios::showbase )
        << "Printing integers preceded by their base:\n"
        << x << '\n'
        << oct << x << '\n'
        << hex << x << endl;
   return 0;
}
```

```
Printing integers preceded by their base:
100
0144
0x64
```

Fig. 11.25 Using the **ios::showbase** flag.

```
// Fig. 11.26: fig11_26.cpp
// Displaying floating-point values in system default,
// scientific, and fixed formats.
#include <iostream.h>

int main()
{
   double x = .001234567, y = 1.946e9;

   cout << "Displayed in default format:\n"
        << x << '\t' << y << '\n';
   cout.setf( ios::scientific, ios::floatfield );
   cout << "Displayed in scientific format:\n"
        << x << '\t' << y << '\n';
   cout.unsetf( ios::scientific );
   cout << "Displayed in default format after unsetf:\n"
        << x << '\t' << y << '\n';
   cout.setf( ios::fixed, ios::floatfield );
   cout << "Displayed in fixed format:\n"
        << x << '\t' << y << endl;
   return 0;
}
```

```
Displayed in default format:
0.00123457      1.946e+009
Displayed in scientific format:
1.234567e-003   1.946000e+009
Displayed in default format after unsetf:
0.00123457      1.946e+009
Displayed in fixed format:
0.001235        1946000000.000000
```

Fig. 11.26 Displaying floating-point values in system default, scientific, and fixed formats.

```
// Fig. 11.27: fig11_27.cpp
// Using the ios::uppercase flag
#include <iostream.h>
#include <iomanip.h>
```

Fig. 11.27 Using the **ios::uppercase** flag (part 1 of 2).

```

int main()
{
   cout << setiosflags( ios::uppercase )
        << "Printing uppercase letters in scientific\n"
        << "notation exponents and hexadecimal values:\n"
        << 4.345e10 << '\n' << hex << 123456789 << endl;
   return 0;
}
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
75BCD15
```

Fig. 11.27 Using the **ios::uppercase** flag (part 2 of 2).

```
// Fig. 11.28: fig11_28.cpp
// Demonstrating the flags member function.
#include <iostream.h>

int main()
{
   int i = 1000;
   double d = 0.0947628;

   cout << "The value of the flags variable is: "
        << cout.flags()
        << "\nPrint int and double in original format:\n"
        << i << '\t' << d << "\n\n";
   long originalFormat =
           cout.flags( ios::oct | ios::scientific );
```

Fig. 11.28 Demonstrating the **flags** member function (part 1 of 2).

```
   cout << "The value of the flags variable is: "
        << cout.flags()
        << "\nPrint int and double in a new format\n"
        << "specified using the flags member function:\n"
        << i << '\t' << d << "\n\n";
   cout.flags( originalFormat );
   cout << "The value of the flags variable is: "
        << cout.flags()
        << "\nPrint values in original format again:\n"
        << i << '\t' << d << endl;
   return 0;
}
```

```
The value of the flags variable is: 0
Print int and double in original format:
1000    0.0947628

The value of the flags variable is: 4040
Print int and double in a new format
specified using the flags member function:
1750    9.476280e-002

The value of the flags variable is: 0
Print values in original format again:
1000    0.0947628
```

Fig. 11.28 Demonstrating the **flags** member function (part 2 of 2).

```
// Fig. 11.29: fig11_29.cpp
// Testing error states.
#include <iostream.h>

int main()
{
   int x;
   cout << "Before a bad input operation:"
        << "\ncin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good()
        << "\n\nExpects an integer, but enter a character: ";
   cin >> x;

   cout << "\nEnter a bad input operation:"
        << "\ncin.rdstate(): " << cin.rdstate()
        << "\n    cin.eof(): " << cin.eof()
        << "\n   cin.fail(): " << cin.fail()
        << "\n    cin.bad(): " << cin.bad()
        << "\n   cin.good(): " << cin.good() << "\n\n";

   cin.clear();

   cout << "After cin.clear()"
        << "\ncin.fail(): " << cin.fail()
        << "\ncin.good(): " << cin.good() << endl;
   return 0;
}
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 2
    cin.bad(): 0
   cin.good(): 0

After cin.clear()
cin.fail(): 0
cin.good(): 1
```

Fig. 11.29 Testing error states.

Illustrations List *(Main Page)*

```
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
}
```

Fig. 12.1 A function template.

```
// Fig 12.2: fig12_02.cpp
// Using template functions
#include <iostream.h>

template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
}

int main()
{
   const int aCount = 5, bCount = 7, cCount = 6;
   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO";  // 6th position for null

   cout << "Array a contains:" << endl;
   printArray( a, aCount );  // integer template function

   cout << "Array b contains:" << endl;
   printArray( b, bCount );  // double template function

   cout << "Array c contains:" << endl;
   printArray( c, cCount );  // character template function

   return 0;
}
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 12.2 Using template functions.

```
// Fig. 12.3: tstack1.h
// Class template Stack
#ifndef TSTACK1_H
#define TSTACK1_H

#include <iostream.h>

template< class T >
class Stack {
public:
   Stack( int = 10 );    // default constructor (stack size 10)
   ~Stack() { delete [] stackPtr; } // destructor
   bool push( const T& ); // push an element onto the stack
   bool pop( T& );        // pop an element off the stack
```

Fig. 12.3 Demonstrating class template **Stack** (part 1 of 4).

```
private:
   int size;             // # of elements in the stack
   int top;              // location of the top element
   T *stackPtr;          // pointer to the stack

   bool isEmpty() const { return top == -1; }      // utility
   bool isFull() const { return top == size - 1; } // functions
};

// Constructor with default size 10
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;               // Stack is initially empty
   stackPtr = new T[ size ]; // allocate space for elements
}

// Push an element onto the stack
// return true if successful, false otherwise
template< class T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue; // place item in Stack
      return true;  // push successful
   }
   return false;     // push unsuccessful
}

// Pop an element off the stack
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // remove item from Stack
      return true;  // pop successful
   }
   return false;     // pop unsuccessful
}

#endif
```

Fig. 12.3 Demonstrating class template **Stack** (part 2 of 4).

```
// Fig. 12.3: fig12_03.cpp
// Test driver for Stack template
#include <iostream.h>
#include "tstack1.h"

int main()
{
   Stack< double > doubleStack( 5 );
   double f = 1.1;
   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( f ) ) { // success true returned
      cout << f << ' ';
      f += 1.1;
   }

   cout << "\nStack is full. Cannot push " << f
        << "\n\nPopping elements from doubleStack\n";

   while ( doubleStack.pop( f ) )  // success true returned
      cout << f << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack;
   int i = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( i ) ) { // success true returned
      cout << i << ' ';
      ++i;
   }

   cout << "\nStack is full. Cannot push " << i
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( i ) )  // success true returned
      cout << i << ' ';

   cout << "\nStack is empty. Cannot pop\n";
   return 0;
}
```

Fig. 12.3 Demonstrating class template **Stack** (part 3 of 4).

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 12.3 Driver for class template **`Stack`** (part 4 of 4).

```cpp
// Fig. 12.4: fig12_04.cpp
// Test driver for Stack template.
// Function main uses a function template to manipulate
// objects of type Stack< T >.
#include <iostream.h>
#include "tstack1.h"

// Function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,    // reference to the Stack< T >
   T value,                 // initial value to be pushed
   T increment,             // increment for subsequent values
   const char *stackName )  // name of the Stack< T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) { // success true returned
      cout << value << ' ';
      value += increment;
   }

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )  // success true returned
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";
}
```

Fig. 12.4 Passing a **`Stack`** template object to a function template (part 1 of 2).

```
int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;
}
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 12.4 Passing a **Stack** template object to a function template (part 2 of 2).

Illustrations List *(Main Page)*

```
// Fig. 13.1: fig13_01.cpp
// A simple exception handling example.
// Checking for a divide-by-zero exception.
#include <iostream.h>

// Class DivideByZeroException to be used in exception
// handling for throwing an exception on a division by zero.
class DivideByZeroException {
public:
   DivideByZeroException()
      : message( "attempted to divide by zero" ) { }
   const char *what() const { return message; }
private:
   const char *message;
};

// Definition of function quotient. Demonstrates throwing
// an exception when a divide-by-zero exception is encountered.
double quotient( int numerator, int denominator )
{
   if ( denominator == 0 )
      throw DivideByZeroException();

   return static_cast< double > ( numerator ) / denominator;
}

// Driver program
int main()
{
   int number1, number2;
   double result;

   cout << "Enter two integers (end-of-file to end): ";

   while ( cin >> number1 >> number2 ) {

      // the try block wraps the code that may throw an
      // exception and the code that should not execute
      // if an exception occurs
      try {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      }
```

Fig. 13.1 A simple exception-handling example with divide by zero (part 1 of 2).

```
      catch ( DivideByZeroException ex ) { // exception handler
         cout << "Exception occurred: " << ex.what() << '\n';
      }

      cout << "\nEnter two integers (end-of-file to end): ";
   }

   cout << endl;
   return 0;      // terminate normally
}
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): 33 9
The quotient is: 3.66667

Enter two integers (end-of-file to end):
```

Fig. 13.1 A simple exception-handling example with divide by zero (part 2 of 2).

```
// Fig. 13.2: fig13_02.cpp
// Demonstration of rethrowing an exception.
#include <iostream>
#include <exception>

using namespace std;

void throwException() throw ( exception )
{
   // Throw an exception and immediately catch it.
   try {
      cout << "Function throwException\n";
      throw exception();  // generate exception
   }
   catch( exception e )
   {
      cout << "Exception handled in function throwException\n";
      throw;  // rethrow exception for further processing
   }
```

Fig. 13.2 Rethrowing an exception (part 1 of 2).

```

   cout << "This also should not print\n";
}

int main()
{
   try {
      throwException();
      cout << "This should not print\n";
   }
   catch ( exception e )
   {
      cout << "Exception handled in main\n";
   }

   cout << "Program control continues after catch in main"
        << endl;
   return 0;
}
```

```
Function throwException
Exception handled in function throwException
Exception handled in main
Program control continues after catch in main
```

Fig. 13.2 Rethrowing an exception (part 2 of 2).

```
// Fig. 13.3: fig13_03.cpp
// Demonstrating stack unwinding.
#include <iostream>
#include <stdexcept>

using namespace std;

void function3() throw ( runtime_error )
{
   throw runtime_error( "runtime_error in function3" );
}

void function2() throw ( runtime_error )
{
   function3();
}

void function1() throw ( runtime_error )
{
   function2();
}

int main()
{
   try {
      function1();
   }
   catch ( runtime_error e )
   {
      cout << "Exception occurred: " << e.what() << endl;
   }

   return 0;
}
```

```
Exception occurred: runtime_error in function3
```

Fig. 13.3 Demonstration of stack unwinding.

```
// Fig. 13.4: fig13_04.cpp
// Demonstrating new returning 0
// when memory is not allocated
#include <iostream.h>

int main()
{
   double *ptr[ 10 ];

   for ( int i = 0; i < 10; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      if ( ptr[ i ] == 0 ) { // new failed to allocate memory
         cout << "Memory allocation failed for ptr[ "
              << i << " ]\n";
         break;
      }
      else
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
   }

   return 0;
}
```

Fig. 13.4 Demonstrating **new** returning 0 on failure (part 1 of 2).

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Memory allocation failed for ptr[ 2 ]
```

Fig. 13.4 Demonstrating **new** returning 0 on failure (part 2 of 2).

```
// Fig. 13.5: fig13_05.cpp
// Demonstrating new throwing bad_alloc
// when memory is not allocated
#include <iostream>
#include <new>

int main()
{
   double *ptr[ 10 ];

   try {
      for ( int i = 0; i < 10; i++ ) {
         ptr[ i ] = new double[ 5000000 ];
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
      }
   }
   catch ( bad_alloc exception ) {
      cout << "Exception occurred: "
           << exception.what() << endl;
   }

   return 0;
}
```

Fig. 13.5 Demonstrating **new** throwing **bad_alloc** on failure (part 1 of 2).

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Exception occurred: Allocation Failure
```

Fig. 13.5 Demonstrating **new** throwing **bad_alloc** on failure (part 2 of 2).

```
// Fig. 13.6: fig13_06.cpp
// Demonstrating set_new_handler
#include <iostream.h>
#include <new.h>
#include <stdlib.h>

void customNewHandler()
{
   cerr << "customNewHandler was called";
   abort();
}

int main()
{
   double *ptr[ 10 ];
   set_new_handler( customNewHandler );

   for ( int i = 0; i < 10; i++ ) {
      ptr[ i ] = new double[ 5000000 ];

      cout << "Allocated 5000000 doubles in ptr[ "
           << i << " ]\n";
   }

   return 0;
}
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
customNewHandler was called
```

Fig. 13.6 Demonstrating **set_new_handler**.

```
// Fig. 13.7: fig13_07.cpp
// Demonstrating auto_ptr
#include <iostream>
#include <memory>

using namespace std;

class Integer {
public:
   Integer( int i = 0 ) : value( i )
      { cout << "Constructor for Integer " << value << endl; }
   ~Integer()
      { cout << "Destructor for Integer " << value << endl; }
   void setInteger( int i ) { value = i; }
   int getInteger() const { return value; }
private:
   int value;
};

int main()
{
   cout << "Creating an auto_ptr object that points "
        << "to an Integer\n";

   auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );

   cout << "Using the auto_ptr to manipulate the Integer\n";
   ptrToInteger->setInteger( 99 );
   cout << "Integer after setInteger: "
        << ( *ptrToInteger ).getInteger()
        << "\nTerminating program" << endl;

   return 0;
}
```

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7
Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99
Terminating program
Destructor for Integer 99
```

Fig. 13.7 Demonstrating `auto_ptr`.

Ilustrations List (Main Page)

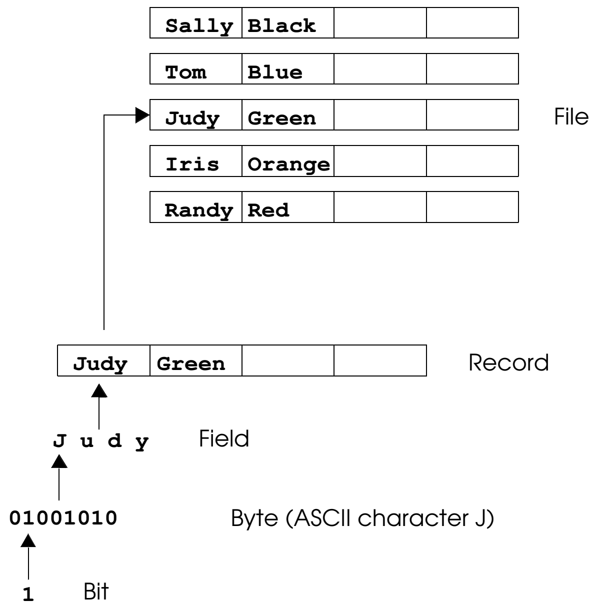


Fig. 14.1 The data hierarchy.

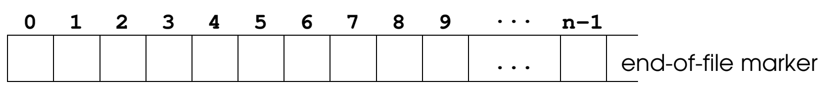


Fig. 14.2 C++'s view of a file of ***n*** bytes.

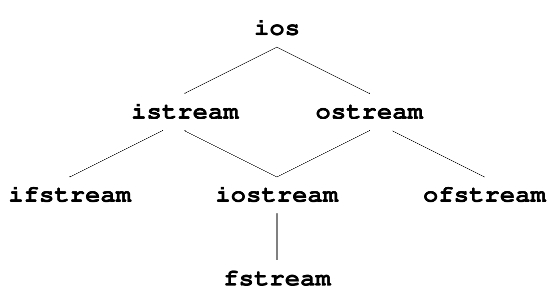


Fig. 14.3 Portion of stream I/O class hierarchy.

```
// Fig. 14.4: fig14_04.cpp
// Create a sequential file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main()
{
   // ofstream constructor opens file
   ofstream outClientFile( "clients.dat", ios::out );

   if ( !outClientFile ) {  // overloaded ! operator
      cerr << "File could not be opened" << endl;
      exit( 1 );    // prototype in stdlib.h
   }

   cout << "Enter the account, name, and balance.\n"
        << "Enter end-of-file to end input.\n? ";

   int account;
   char name[ 30 ];
   float balance;

   while ( cin >> account >> name >> balance ) {
      outClientFile << account << ' ' << name
                    << ' ' << balance << '\n';
      cout << "? ";
   }

   return 0;  // ofstream destructor closes file
}
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 14.4 Creating a sequential file.

Mode	Description
`ios::app`	Write all output to the end of the file.
`ios::ate`	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
`ios::in`	Open a file for input.
`ios::out`	Open a file for output.
`ios::trunc`	Discard the file's contents if it exists (this is also the default action for `ios::out`)
`ios::nocreate`	If the file does not exist, the open operation fails.
`ios::noreplace`	If the file exists, the open operation fails.

Fig. 14.5 File open modes.

Computer system	Keyboard combination
UNIX systems	*<ctrl> d* (on a line by itself)
IBM PC and compatibles	*<ctrl> z*
Macintosh	*<ctrl> d*
VAX (VMS)	*<ctrl> z*

Fig. 14.6 End-of-file key combinations for various popular computer systems.

```
// Fig. 14.7: fig14_07.cpp
// Reading and printing a sequential file
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

void outputLine( int, const char *, double );

int main()
{
   // ifstream constructor opens the file
   ifstream inClientFile( "clients.dat", ios::in );

   if ( !inClientFile ) {
      cerr << "File could not be opened\n";
      exit( 1 );
   }

   int account;
   char name[ 30 ];
   double balance;

   cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
```

```
         << setw( 13 ) << "Name" << "Balance\n";

   while ( inClientFile >> account >> name >> balance )
      outputLine( account, name, balance );

   return 0;  // ifstream destructor closes the file
}
```

Fig. 14.7 Reading and printing a sequential file (part 1 of 2).

```

void outputLine( int acct, const char *name, double bal )
{
   cout << setiosflags( ios::left ) << setw( 10 ) << acct
        << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
        << resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )
        << bal << '\n';
}
```

```
Account   Name          Balance
100       Jones           24.98
200       Doe            345.67
300       White            0.00
400       Stone          -42.16
500       Rich           224.62
```

Fig. 14.7 Reading and printing a sequential file (part 2 of 2).

```
// Fig. 14.8: fig14_08.cpp
// Credit inquiry program
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE,
                   DEBIT_BALANCE, END };
int getRequest();
bool shouldDisplay( int, double );
void outputLine( int, const char *, double );

int main()
{
   // ifstream constructor opens the file
   ifstream inClientFile( "clients.dat", ios::in );

   if ( !inClientFile ) {
      cerr << "File could not be opened" << endl;
      exit( 1 );
   }

   int request;
   int account;
   char name[ 30 ];
   double balance;

   cout << "Enter request\n"
        << " 1 - List accounts with zero balances\n"
        << " 2 - List accounts with credit balances\n"
```

```
          << " 3 - List accounts with debit balances\n"
          << " 4 - End of run";
   request = getRequest();

   while ( request != END ) {

      switch ( request ) {
         case ZERO_BALANCE:
            cout << "\nAccounts with zero balances:\n";
            break;
         case CREDIT_BALANCE:
            cout << "\nAccounts with credit balances:\n";
            break;
         case DEBIT_BALANCE:
            cout << "\nAccounts with debit balances:\n";
            break;
      }

      inClientFile >> account >> name >> balance;

```

Fig. 14.8 Credit inquiry program (part 1 of 2).

```
      while ( !inClientFile.eof() ) {
         if ( shouldDisplay( request, balance ) )
            outputLine( account, name, balance );

         inClientFile >> account >> name >> balance;
      }

      inClientFile.clear();    // reset eof for next input
      inClientFile.seekg( 0 ); // move to beginning of file
      request = getRequest();
   }

   cout << "End of run." << endl;

   return 0;   // ifstream destructor closes the file
}

int getRequest()
{
   int request;

   do {
      cout << "\n? ";
      cin >> request;
   } while( request < ZERO_BALANCE && request > END );

   return request;
}

bool shouldDisplay( int type, double balance )
{
   if ( type == CREDIT_BALANCE && balance < 0 )
      return true;

   if ( type == DEBIT_BALANCE && balance > 0 )
      return true;

   if ( type == ZERO_BALANCE && balance == 0 )
      return true;

   return false;
```

```
}

void outputLine( int acct, const char *name, double bal )
{
   cout << setiosflags( ios::left ) << setw( 10 ) << acct
        << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
        << resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )
        << bal << '\n';
}
```

Fig. 14.8 Credit inquiry program (part 2 of 2).

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

? 2

Accounts with credit balances:
400       Stone         -42.16

? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

? 4
End of run.
```

Fig. 14.9 Sample output of the credit inquiry program of Fig. 14.8.

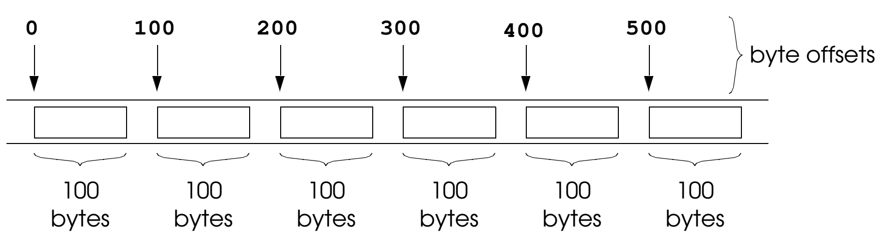


Fig. 14.10 C++'s view of a random access file.

```
// Fig. 14.11: clntdata.h
// Definition of struct clientData used in
// Figs. 14.11, 14.12, 14.14 and 14.15.
#ifndef CLNTDATA_H
#define CLNTDATA_H

struct clientData {
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   float balance;
};

#endif
```

Fig. 14.11 Creating a random access file sequentially (part 1 of 2).

```
// Fig. 14.11: fig14_11.cpp
// Creating a randomly accessed file sequentially
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"

int main()
{
   ofstream outCredit( "credit.dat", ios::out );

   if ( !outCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }

   clientData blankClient = { 0, "", "", 0.0 };

   for ( int i = 0; i < 100; i++ )
      outCredit.write(
         reinterpret_cast<const char *>( &blankClient ),
         sizeof( clientData ) );
   return 0;
}
```

Fig. 14.11 Creating a random access file sequentially (part 2 of 2).

```
// Fig. 14.12: fig14_12.cpp
// Writing to a random access file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"

int main()
{
   ofstream outCredit( "credit.dat", ios::ate );

   if ( !outCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }

```

```
   cout << "Enter account number "
        << "(1 to 100, 0 to end input)\n? ";

   clientData client;
   cin >> client.accountNumber;

   while ( client.accountNumber > 0 &&
           client.accountNumber <= 100 ) {
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> client.lastName >> client.firstName
          >> client.balance;

      outCredit.seekp( ( client.accountNumber - 1 ) *
                       sizeof( clientData ) );
      outCredit.write(
         reinterpret_cast<const char *>( &client ),
         sizeof( clientData ) );

```

Fig. 14.12 Writing data randomly to a random access file (part 1 of 2).

```
      cout << "Enter account number\n? ";
      cin >> client.accountNumber;
   }

   return 0;
}
```

Fig. 14.12 Writing data randomly to a random access file (part 2 of 2).

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Fig. 14.13 Sample execution of the program in Fig. 14.12.

```cpp
// Fig. 14.14: fig14_14.cpp
// Reading a random access file sequentially
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"

void outputLine( ostream&, const clientData & );

int main()
{
   ifstream inCredit( "credit.dat", ios::in );

   if ( !inCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }

   cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
        << setw( 16 ) << "Last Name" << setw( 11 )
        << "First Name" << resetiosflags( ios::left )
        << setw( 10 ) << "Balance" << endl;

   clientData client;

   inCredit.read( reinterpret_cast<char *>( &client ),
                  sizeof( clientData ) );

```

Fig. 14.14 Reading a random access file sequentially (part 1 of 2).

```cpp
   while ( inCredit && !inCredit.eof() ) {

      if ( client.accountNumber != 0 )
         outputLine( cout, client );

      inCredit.read( reinterpret_cast<char *>( &client ),
                     sizeof( clientData ) );
   }

   return 0;
}

void outputLine( ostream &output, const clientData &c )
{
   output << setiosflags( ios::left ) << setw( 10 )
          << c.accountNumber << setw( 16 ) << c.lastName
          << setw( 11 ) << c.firstName << setw( 10 )
          << setprecision( 2 ) << resetiosflags( ios::left )
          << setiosflags( ios::fixed | ios::showpoint )
          << c.balance << '\n';
```

```
}
```

```
Account   Last Name        First Name    Balance
29        Brown            Nancy          -24.54
33        Dunn             Stacey         314.33
37        Barker           Doug             0.00
88        Smith            Dave           258.34
96        Stone            Sam             34.98
```

Fig. 14.14 Reading a random access file sequentially (part 2 of 2).

```
// Fig. 14.15: fig14_15.cpp
// This program reads a random access file sequentially,
// updates data already written to the file, creates new
// data to be placed in the file, and deletes data
// already in the file.
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include "clntdata.h"

int enterChoice();
void textFile( fstream& );
void updateRecord( fstream& );
void newRecord( fstream& );
void deleteRecord( fstream& );
void outputLine( ostream&, const clientData & );
```

Fig. 14.15 Bank account program (part 1 of 5).

```
int getAccount( const char * );

enum Choices { TEXTFILE = 1, UPDATE, NEW, DELETE, END };

int main()
{
   fstream inOutCredit( "credit.dat", ios::in | ios::out );

   if ( !inOutCredit ) {
      cerr << "File could not be opened." << endl;
      exit ( 1 );
   }

   int choice;

   while ( ( choice = enterChoice() ) != END ) {

      switch ( choice ) {
         case TEXTFILE:
            textFile( inOutCredit );
            break;
         case UPDATE:
            updateRecord( inOutCredit );
            break;
         case NEW:
            newRecord( inOutCredit );
            break;
         case DELETE:
```

```
            deleteRecord( inOutCredit );
            break;
         default:
            cerr << "Incorrect choice\n";
            break;
      }

      inOutCredit.clear();  // resets end-of-file indicator
   }

   return 0;
}

// Prompt for and input menu choice
int enterChoice()
{
   cout << "\nEnter your choice" << endl
        << "1 - store a formatted text file of accounts\n"
        << "    called \"print.txt\" for printing\n"
        << "2 - update an account\n"
        << "3 - add a new account\n"
        << "4 - delete an account\n"
        << "5 - end program\n? ";
```

Fig. 14.15 Bank account program (part 2 of 5).

```

   int menuChoice;
   cin >> menuChoice;
   return menuChoice;
}

// Create formatted text file for printing
void textFile( fstream &readFromFile )
{
   ofstream outPrintFile( "print.txt", ios::out );

   if ( !outPrintFile ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }

   outPrintFile << setiosflags( ios::left ) << setw( 10 )
       << "Account" << setw( 16 ) << "Last Name" << setw( 11 )
       << "First Name" << resetiosflags( ios::left )
       << setw( 10 ) << "Balance" << endl;
   readFromFile.seekg( 0 );

   clientData client;
   readFromFile.read( reinterpret_cast<char *>( &client ),
                      sizeof( clientData ) );

   while ( !readFromFile.eof() ) {
      if ( client.accountNumber != 0 )
         outputLine( outPrintFile, client );

      readFromFile.read( reinterpret_cast<char *>( &client ),
                         sizeof( clientData ) );
   }
}

// Update an account's balance
void updateRecord( fstream &updateFile )
{
```

```
   int account = getAccount( "Enter account to update" );

   updateFile.seekg( ( account - 1 ) * sizeof( clientData ) );

   clientData client;
   updateFile.read( reinterpret_cast<char *>( &client ),
                    sizeof( clientData ) );

   if ( client.accountNumber != 0 ) {
      outputLine( cout, client );
      cout << "\nEnter charge (+) or payment (-): ";

      float transaction;    // charge or payment
```

Fig. 14.15 Bank account program (part 3 of 5).

```
      cin >> transaction;   // should validate
      client.balance += transaction;
      outputLine( cout, client );
      updateFile.seekp( ( account-1 ) * sizeof( clientData ) );
      updateFile.write(
         reinterpret_cast<const char *>( &client ),
         sizeof( clientData ) );
   }
   else
      cerr << "Account #" << account
           << " has no information." << endl;
}

// Create and insert new record
void newRecord( fstream &insertInFile )
{
   int account = getAccount( "Enter new account number" );

   insertInFile.seekg( ( account-1 ) * sizeof( clientData ) );

   clientData client;
   insertInFile.read( reinterpret_cast<char *>( &client ),
                      sizeof( clientData ) );

   if ( client.accountNumber == 0 ) {
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> client.lastName >> client.firstName
          >> client.balance;
      client.accountNumber = account;
      insertInFile.seekp( ( account - 1 ) *
                          sizeof( clientData ) );
      insertInFile.write(
         reinterpret_cast<const char *>( &client ),
         sizeof( clientData ) );
   }
   else
      cerr << "Account #" << account
           << " already contains information." << endl;
}

// Delete an existing record
void deleteRecord( fstream &deleteFromFile )
{
   int account = getAccount( "Enter account to delete" );

   deleteFromFile.seekg( (account-1) * sizeof( clientData ) );

   clientData client;
```

Illustrations List ***(Main Page)***

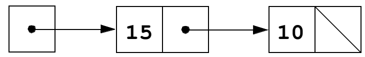


Fig. 15.1 Two self-referential class objects linked together.

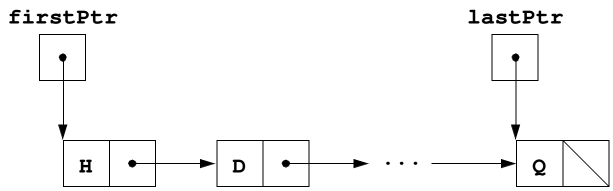


Fig. 15.2 A graphical representation of a list.

```
// Fig. 15.3: listnd.h
// ListNode template definition
#ifndef LISTND_H
#define LISTND_H
```

Fig. 15.3 Manipulating a linked list (part 1 of 8).

```

template< class NODETYPE > class List;  // forward declaration

template<class NODETYPE>
class ListNode {
   friend class List< NODETYPE >; // make List a friend
public:
   ListNode( const NODETYPE & );  // constructor
   NODETYPE getData() const;      // return data in the node
private:
   NODETYPE data;                 // data
   ListNode< NODETYPE > *nextPtr; // next node in the list
};

// Constructor
template<class NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 ) { }

// Return a copy of the data in the node
template< class NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const { return data; }

#endif
```

Fig. 15.3 Manipulating a linked list (part 2 of 8).

```
// Fig. 15.3: list.h
// Template List class definition
#ifndef LIST_H
#define LIST_H

#include <iostream.h>
#include <assert.h>
#include "listnd.h"

template< class NODETYPE >
class List {
```

```
public:
   List();      // constructor
   ~List();     // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
private:
   ListNode< NODETYPE > *firstPtr;  // pointer to first node
   ListNode< NODETYPE > *lastPtr;   // pointer to last node
```

Fig. 15.3 Manipulating a linked list (part 3 of 8).

```

   // Utility function to allocate a new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
};

// Default constructor
template< class NODETYPE >
List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) { }

// Destructor
template< class NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) {    // List is not empty
      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;

      while ( currentPtr != 0 ) {  // delete remaining nodes
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      }
   }

   cout << "All nodes destroyed\n\n";
}

// Insert a node at the front of the list
template< class NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;
   else {           // List is not empty
      newPtr->nextPtr = firstPtr;
      firstPtr = newPtr;
   }
}

// Insert a node at the back of the list
template< class NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

```

```
   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;
```

Fig. 15.3 Manipulating a linked list (part 4 of 8).

```
   else {           // List is not empty
      lastPtr->nextPtr = newPtr;
      lastPtr = newPtr;
   }
}

// Delete a node from the front of the list
template< class NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() )              // List is empty
      return false;              // delete unsuccessful
   else {
      ListNode< NODETYPE > *tempPtr = firstPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else
         firstPtr = firstPtr->nextPtr;

      value = tempPtr->data;  // data being removed
      delete tempPtr;
      return true;            // delete successful
   }
}

// Delete a node from the back of the list
template< class NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() )
      return false;   // delete unsuccessful
   else {
      ListNode< NODETYPE > *tempPtr = lastPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr;

         lastPtr = currentPtr;
         currentPtr->nextPtr = 0;
      }

      value = tempPtr->data;
      delete tempPtr;
```

Fig. 15.3 Manipulating a linked list (part 5 of 8).

```
      return true;   // delete successful
   }
}

// Is the List empty?
```

```
template< class NODETYPE >
bool List< NODETYPE >::isEmpty() const
   { return firstPtr == 0; }

// Return a pointer to a newly allocated node
template< class NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
                                        const NODETYPE &value )
{
   ListNode< NODETYPE > *ptr =
      new ListNode< NODETYPE >( value );
   assert( ptr != 0 );
   return ptr;
}

// Display the contents of the List
template< class NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) {
      cout << "The list is empty\n\n";
      return;
   }

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
   }

   cout << "\n\n";
}

#endif
```

Fig. 15.3 Manipulating a linked list (part 6 of 8).

```
// Fig. 15.3: fig15_03.cpp
// List class test
#include <iostream.h>
#include "list.h"

// Function to test an integer List
template< class T >
void testList( List< T > &listObject, const char *type )
{
   cout << "Testing a List of " << type << " values\n";

   instructions();
   int choice;
   T value;

   do {
      cout << "? ";
      cin >> choice;

      switch ( choice ) {
         case 1:
            cout << "Enter " << type << ": ";
            cin >> value;
            listObject.insertAtFront( value );
```

```
            listObject.print();
            break;
         case 2:
            cout << "Enter " << type << ": ";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3:
            if ( listObject.removeFromFront( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
         case 4:
            if ( listObject.removeFromBack( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
      }
   } while ( choice != 5 );

   cout << "End list test\n\n";
}

```

Fig. 15.3 Manipulating a linked list (part 7 of 8).

```
void instructions()
{
   cout << "Enter one of the following:\n"
        << "  1 to insert at beginning of list\n"
        << "  2 to insert at end of list\n"
        << "  3 to delete from beginning of list\n"
        << "  4 to delete from end of list\n"
        << "  5 to end list processing\n";
}

int main()
{
   List< int > integerList;
   testList( integerList, "integer" ); // test integerList

   List< float > floatList;
   testList( floatList, "float" );     // test integerList

   return 0;
}
```

Fig. 15.3 Manipulating a linked list (part 8 of 8).

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test
```

Fig. 15.4 Sample output for the program of Fig. 15.3 (part 1 of 2).

```
Testing a List of float values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter float: 1.1
The list is: 1.1

? 1
Enter float: 2.2
The list is: 2.2 1.1

? 2
Enter float: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter float: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Fig. 15.4 Sample output for the program of Fig. 15.3 (part 2 of 2).

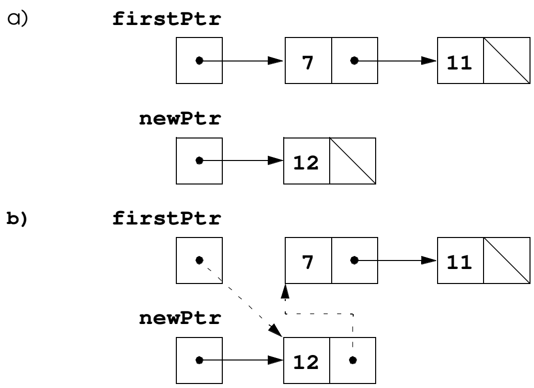


Fig. 15.5 The **insertAtFront** operation.

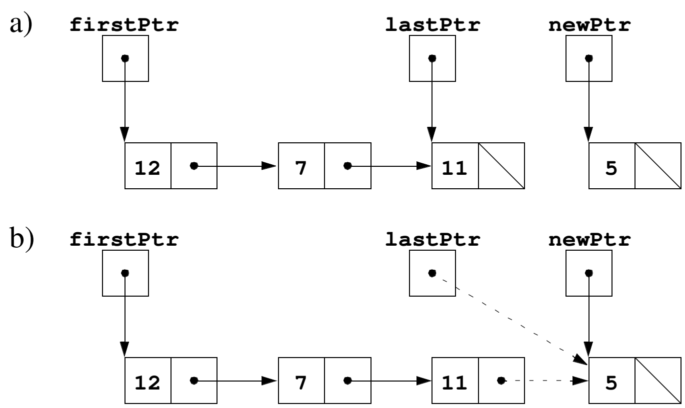


Fig. 15.6 A graphical representation of the **insertAtBack** operation.

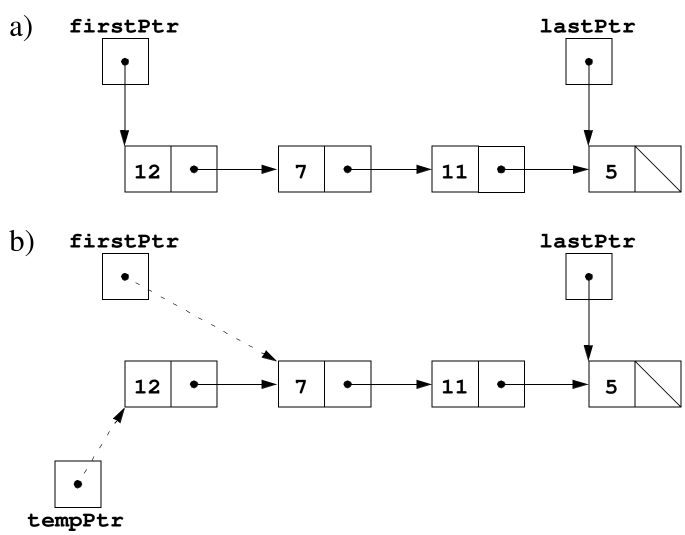


Fig. 15.7 A graphical representation of the **removeFromFront** operation.

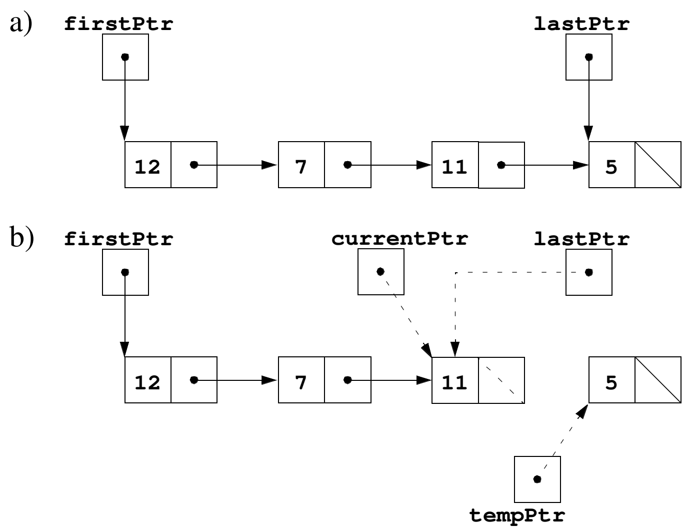


Fig. 15.8 A graphical representation of the **`removeFromBack`** operation.

```
// Fig. 15.9: stack.h
// Stack class template definition
// Derived from class List
#ifndef STACK_H
#define STACK_H

#include "list.h"

template< class STACKTYPE >
class Stack : private List< STACKTYPE > {
```

Fig. 15.9 A simple stack program (part 1 of 3).

```
public:
   void push( const STACKTYPE &d ) { insertAtFront( d ); }
   bool pop( STACKTYPE &d ) { return removeFromFront( d ); }
   bool isStackEmpty() const { return isEmpty(); }
   void printStack() const { print(); }
};

#endif
```

Fig. 15.9 A simple stack program (part 2 of 3).

```
// Fig. 15.9: fig15_09.cpp
// Driver to test the template Stack class
#include <iostream.h>
#include "stack.h"

int main()
{
   Stack< int > intStack;
   int popInteger;
   cout << "processing an integer Stack" << endl;

   for ( int i = 0; i < 4; i++ ) {
      intStack.push( i );
```

```
      intStack.printStack();
   }

   while ( !intStack.isStackEmpty() ) {
      intStack.pop( popInteger );
      cout << popInteger << " popped from stack" << endl;
      intStack.printStack();
   }

   Stack< double > doubleStack;
   double val = 1.1, popdouble;
   cout << "processing a double Stack" << endl;

   for ( i = 0; i < 4; i++ ) {
      doubleStack.push( val );
      doubleStack.printStack();
      val += 1.1;
   }

   while ( !doubleStack.isStackEmpty() ) {
      doubleStack.pop( popdouble );
      cout << popdouble << " popped from stack" << endl;
      doubleStack.printStack();
   }
   return 0;
}
```

Fig. 15.9 A simple stack program (part 3 of 3).

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

The list is: 3 2 1 0

3 popped from stack
The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

The list is: 4.4 3.3 2.2 1.1

4.4 popped from stack
The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 15.10 Sample output from the program of Fig. 15.9.

```
// Fig. 15.11: stack_c.h
// Definition of Stack class composed of List object
#ifndef STACK_C
#define STACK_C
#include "list.h"

template< class STACKTYPE >
class Stack {
public:
   // no constructor; List constructor does initialization
```

```
   void push( const STACKTYPE &d ) { s.insertAtFront( d ); }
   bool pop( STACKTYPE &d ) { return s.removeFromFront( d ); }
   bool isStackEmpty() const { return s.isEmpty(); }
   void printStack() const { s.print(); }
private:
   List< STACKTYPE > s;
};

#endif
```

Fig. 15.11 A simple stack program using composition.

```
// Fig. 15.12: queue.h
// Queue class template definition
// Derived from class List
#ifndef QUEUE_H
#define QUEUE_H

#include "list.h"

template< class QUEUETYPE >
class Queue: private List< QUEUETYPE > {
public:
   void enqueue( const QUEUETYPE &d ) { insertAtBack( d ); }
   bool dequeue( QUEUETYPE &d )
      { return removeFromFront( d ); }
   bool isQueueEmpty() const { return isEmpty(); }
   void printQueue() const { print(); }
};

#endif
```

Fig. 15.12 Processing a queue (part 1 of 2).

```
// Fig. 15.12: fig15_12.cpp
// Driver to test the template Queue class
#include <iostream.h>
#include "queue.h"

int main()
{
   Queue< int > intQueue;
   int dequeueInteger;
   cout << "processing an integer Queue" << endl;

   for ( int i = 0; i < 4; i++ ) {
      intQueue.enqueue( i );
      intQueue.printQueue();
   }

   while ( !intQueue.isQueueEmpty() ) {
      intQueue.dequeue( dequeueInteger );
      cout << dequeueInteger << " dequeued" << endl;
      intQueue.printQueue();
   }

   Queue< double > doubleQueue;
   double val = 1.1, dequeuedouble;

   cout << "processing a double Queue" << endl;

```

```
    for ( i = 0; i < 4; i++ ) {
       doubleQueue.enqueue( val );
       doubleQueue.printQueue();
       val += 1.1;
    }

    while ( !doubleQueue.isQueueEmpty() ) {
       doubleQueue.dequeue( dequeuedouble );
       cout << dequeuedouble << " dequeued" << endl;
       doubleQueue.printQueue();
    }

    return 0;
}
```

Fig. 15.12 Processing a queue (part 2 of 2).

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

The list is: 0 1 2 3

0 dequeued
The list is: 1 2 3

1 dequeued
The list is: 2 3

2 dequeued
The list is: 3

3 dequeued
The list is empty

processing a float Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

The list is: 1.1 2.2 3.3 4.4

1.1 dequeued
The list is: 2.2 3.3 4.4

2.2 dequeued
The list is: 3.3 4.4

3.3 dequeued
The list is: 4.4

4.4 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 15.13 Sample output from the program in Fig. 15.12.

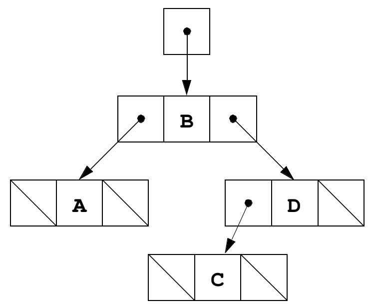


Fig. 15.14 A graphical representation of a binary tree.

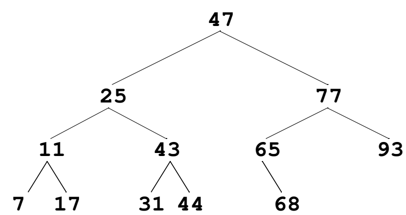


Fig. 15.15 A binary search tree.

```
// Fig. 15.16: treenode.h
// Definition of class TreeNode
#ifndef TREENODE_H
#define TREENODE_H

template< class NODETYPE > class Tree;  // forward declaration

template< class NODETYPE >
class TreeNode {
   friend class Tree< NODETYPE >;
public:
   TreeNode( const NODETYPE &d )
      : leftPtr( 0 ), data( d ), rightPtr( 0 ) { }
   NODETYPE getData() const { return data; }
private:
   TreeNode< NODETYPE > *leftPtr;  // pointer to left subtree
   NODETYPE data;
   TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
};

#endif
```

Fig. 15.16 Creating and traversing a binary tree (part 1 of 6).

```
// Fig. 15.16: fig15_16.cpp
// Definition of template class Tree
#ifndef TREE_H
#define TREE_H

#include <iostream.h>
#include <assert.h>
#include "treenode.h"
```

```
template< class NODETYPE >
class Tree {
public:
   Tree();
   void insertNode( const NODETYPE & );
   void preOrderTraversal() const;
   void inOrderTraversal() const;
   void postOrderTraversal() const;
private:
   TreeNode< NODETYPE > *rootPtr;

   // utility functions
   void insertNodeHelper(
           TreeNode< NODETYPE > **, const NODETYPE & );
   void preOrderHelper( TreeNode< NODETYPE > * ) const;
   void inOrderHelper( TreeNode< NODETYPE > * ) const;
   void postOrderHelper( TreeNode< NODETYPE > * ) const;
};

template< class NODETYPE >
Tree< NODETYPE >::Tree() { rootPtr = 0; }

template< class NODETYPE >
void Tree< NODETYPE >::insertNode( const NODETYPE &value )
   { insertNodeHelper( &rootPtr, value ); }
```

Fig. 15.16 Creating and traversing a binary tree (part 2 of 6).

```

// This function receives a pointer to a pointer so the
// pointer can be modified.
template< class NODETYPE >
void Tree< NODETYPE >::insertNodeHelper(
        TreeNode< NODETYPE > **ptr, const NODETYPE &value )
{
   if ( *ptr == 0 ) {                   // tree is empty
      *ptr = new TreeNode< NODETYPE >( value );
      assert( *ptr != 0 );
   }
   else                              // tree is not empty
      if ( value < ( *ptr )->data )
         insertNodeHelper( &( ( *ptr )->leftPtr ), value );
      else
         if ( value > ( *ptr )->data )
            insertNodeHelper( &( ( *ptr )->rightPtr ), value );
         else
            cout << value << " dup" << endl;
}

template< class NODETYPE >
void Tree< NODETYPE >::preOrderTraversal() const
   { preOrderHelper( rootPtr ); }

template< class NODETYPE >
void Tree< NODETYPE >::preOrderHelper(
                          TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 ) {
      cout << ptr->data << ' ';
      preOrderHelper( ptr->leftPtr );
      preOrderHelper( ptr->rightPtr );
   }
}
```

```
template< class NODETYPE >
void Tree< NODETYPE >::inOrderTraversal() const
   { inOrderHelper( rootPtr ); }

template< class NODETYPE >
void Tree< NODETYPE >::inOrderHelper(
                              TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 ) {
      inOrderHelper( ptr->leftPtr );
      cout << ptr->data << ' ';
      inOrderHelper( ptr->rightPtr );
   }
}
```

Fig. 15.16 Creating and traversing a binary tree (part 3 of 6).

```
template< class NODETYPE >
void Tree< NODETYPE >::postOrderTraversal() const
   { postOrderHelper( rootPtr ); }

template< class NODETYPE >
void Tree< NODETYPE >::postOrderHelper(
                              TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 ) {
      postOrderHelper( ptr->leftPtr );
      postOrderHelper( ptr->rightPtr );
      cout << ptr->data << ' ';
   }
}

#endif
```

Fig. 15.16 Creating and traversing a binary tree (part 4 of 6).

```
// Fig. 15.16: fig15_16.cpp
// Driver to test class Tree
#include <iostream.h>
#include <iomanip.h>
#include "tree.h"

int main()
{
   Tree< int > intTree;
   int intVal;

   cout << "Enter 10 integer values:\n";
   for( int i = 0; i < 10; i++ ) {
      cin >> intVal;
      intTree.insertNode( intVal );
   }

   cout << "\nPreorder traversal\n";
   intTree.preOrderTraversal();

   cout << "\nInorder traversal\n";
   intTree.inOrderTraversal();

   cout << "\nPostorder traversal\n";
```

```
    intTree.postOrderTraversal();

    Tree< double > doubleTree;
    double doubleVal;

```

Fig. 15.16 Creating and traversing a binary tree (part 5 of 6).

```
    cout << "\n\n\nEnter 10 double values:\n"
         << setiosflags( ios::fixed | ios::showpoint )
         << setprecision( 1 );
    for ( i = 0; i < 10; i++ ) {
       cin >> doubleVal;
       doubleTree.insertNode( doubleVal );
    }

    cout << "\nPreorder traversal\n";
    doubleTree.preOrderTraversal();

    cout << "\nInorder traversal\n";
    doubleTree.inOrderTraversal();

    cout << "\nPostorder traversal\n";
    doubleTree.postOrderTraversal();

    return 0;
}
```

Fig. 15.16 Creating and traversing a binary tree (part 6 of 6).

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

Fig. 15.17 Sample output from the program of Fig. 15.16.

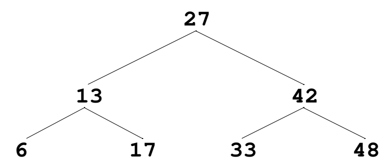


Fig. 15.18 A binary search tree.

Illustrations List *(Main Page)*

Byte	0	1	2	3
	01100001		00000000	01100001

Fig. 16.1 A possible storage alignment for a variable of type **`Example`** showing an undefined area in memory.

```
// Fig. 16.2: fig16_02.cpp
// Card shuffling and dealing program using structures
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

struct Card {
   char *face;
   char *suit;
};

void fillDeck( Card *, char *[], char *[] );
void shuffle( Card * );
void deal( Card * );

int main()
{
   Card deck[ 52 ];
   char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
                    "Six", "Seven", "Eight", "Nine", "Ten",
                    "Jack", "Queen", "King" };
   char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades" };

   srand( time( 0 ) );           // randomize
```

Fig. 16.2 High-performance card shuffling and dealing simulation (part 1 of 2).

```
   fillDeck( deck, face, suit );
   shuffle( deck );
   deal( deck );
   return 0;
}

void fillDeck( Card *wDeck, char *wFace[], char *wSuit[] )
{
   for ( int i = 0; i < 52; i++ ) {
      wDeck[ i ].face = wFace[ i % 13 ];
      wDeck[ i ].suit = wSuit[ i / 13 ];
   }
}

void shuffle( Card *wDeck )
{
   for ( int i = 0; i < 52; i++ ) {
      int j = rand() % 52;
      Card temp = wDeck[ i ];
      wDeck[ i ] = wDeck[ j ];
      wDeck[ j ] = temp;
   }
}

void deal( Card *wDeck )
{
```

```
    for ( int i = 0; i < 52; i++ )
       cout << setiosflags( ios::right )
            << setw( 5 ) << wDeck[ i ].face << " of "
            << setiosflags( ios::left )
            << setw( 8 ) << wDeck[ i ].suit
            << ( ( i + 1 ) % 2 ? '\t' : '\n' );
}
```

Fig. 16.2 High-performance card shuffling and dealing simulation (part 2 of 2).

```
Eight of Diamonds         Ace of Hearts
Eight of Clubs           Five of Spades
Seven of Hearts         Deuce of Diamonds
  Ace of Clubs            Ten of Diamonds
Deuce of Spades           Six of Diamonds
Seven of Spades         Deuce of Clubs
 Jack of Clubs            Ten of Spades
 King of Hearts          Jack of Diamonds
Three of Hearts         Three of Diamonds
Three of Clubs           Nine of Clubs
  Ten of Hearts         Deuce of Hearts
  Ten of Clubs          Seven of Diamonds
  Six of Clubs          Queen of Spades
  Six of Hearts         Three of Spades
 Nine of Diamonds         Ace of Diamonds
 Jack of Spades          Five of Clubs
 King of Diamonds       Seven of Clubs
 Nine of Spades          Four of Hearts
  Six of Spades         Eight of Spades
Queen of Diamonds        Five of Diamonds
  Ace of Spades          Nine of Hearts
 King of Clubs           Five of Hearts
 King of Spades          Four of Diamonds
Queen of Hearts         Eight of Hearts
 Four of Spades          Jack of Hearts
 Four of Clubs          Queen of Clubs
```

Fig. 16.3 Output for the high-performance card shuffling and dealing simulation.

Operator	Name	Description
`&`	bitwise AND	The bits in the result are set to **1** if the corresponding bits in the two operands are both **1**.
`|`	bitwise inclusive OR	The bits in the result are set to **1** if at least one of the corresponding bits in the two operands is **1**.
`^`	bitwise exclusive OR	The bits in the result are set to **1** if exactly one of the corresponding bits in the two operands is **1**.
`<<`	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with **0** bits.
`>>`	right shift with sign extension	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
`~`	one's complement	All **0** bits are set to **1** and all **1** bits are set to **0**.

Fig. 16.4 The bitwise operators .

```
// Fig. 16.5: fig16_05.cpp
// Printing an unsigned integer in bits
#include <iostream.h>
#include <iomanip.h>

void displayBits( unsigned );

int main()
{
   unsigned x;

   cout << "Enter an unsigned integer: ";
   cin >> x;
   displayBits( x );
   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 15;

   cout << setw( 7 ) << value << " = ";

   for ( c = 1; c <= 16; c++ ) {
      cout << ( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         cout << ' ';
   }

   cout << endl;
}
```

```
Enter an unsigned integer: 65000
  65000 = 11111101 11101000
```

Fig. 16.5 Printing an unsigned integer in bits.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 16.6 Results of combining two bits with the bitwise AND operator (&).

```
// Fig. 16.7: fig16_07.cpp
// Using the bitwise AND, bitwise inclusive OR, bitwise
// exclusive OR, and bitwise complement operators.
#include <iostream.h>
#include <iomanip.h>

void displayBits( unsigned );

int main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
```

Fig. 16.7 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and bitwise complement operators (part 1 of 2).

```
   cout << "The result of combining the following\n";
   displayBits( number1 );
   displayBits( mask );
   cout << "using the bitwise AND operator & is\n";
   displayBits( number1 & mask );

   number1 = 15;
   setBits = 241;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( setBits );
   cout << "using the bitwise inclusive OR operator | is\n";
   displayBits( number1 | setBits );

   number1 = 139;
   number2 = 199;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( number2 );
   cout << "using the bitwise exclusive OR operator ^ is\n";
   displayBits( number1 ^ number2 );

```

```
   number1 = 21845;
   cout << "\nThe one's complement of\n";
   displayBits( number1 );
   cout << "is" << endl;
   displayBits( ~number1 );

   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 15;

   cout << setw( 7 ) << value << " = ";

   for ( c = 1; c <= 16; c++ ) {
      cout << ( value & displayMask ? '1' : '0' );
      value <<= 1;

      if ( c % 8 == 0 )
         cout << ' ';
   }

   cout << endl;
}
```

Fig. 16.7 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and bitwise complement operators (part 2 of 2).

```
The result of combining the following
  65535 = 11111111 11111111
      1 = 00000000 00000001
using the bitwise AND operator & is
      1 = 00000000 00000001

The result of combining the following
     15 = 00000000 00001111
    241 = 00000000 11110001
using the bitwise inclusive OR operator | is
    255 = 00000000 11111111

The result of combining the following
    139 = 00000000 10001011
    199 = 00000000 11000111
using the bitwise exclusive OR operator ^ is
     76 = 00000000 01001100

The one's complement of
  21845 = 01010101 01010101
is
  43690 = 10101010 10101010
```

Fig. 16.8 Output for the program of Fig. 16.7.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 16.9 Results of combining two bits with the bitwise inclusive OR operator (**|**).

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 16.10 Results of combining two bits with the bitwise exclusive OR operator (**^**).

```
// Fig. 16.11: fig16_11.cpp
// Using the bitwise shift operators
#include <iostream.h>
#include <iomanip.h>

void displayBits( unsigned );

int main()
{
   unsigned number1 = 960;

   cout << "The result of left shifting\n";
   displayBits( number1 );
   cout << "8 bit positions using the left "
        << "shift operator is\n";
```

Fig. 16.11 Using the bitwise shift operators (part 1 of 2).

```
   displayBits( number1 << 8 );
   cout << "\nThe result of right shifting\n";
   displayBits( number1 );
   cout << "8 bit positions using the right "
        << "shift operator is\n";
   displayBits( number1 >> 8 );
   return 0;
}

void displayBits( unsigned value )
{
   unsigned c, displayMask = 1 << 15;

   cout << setw( 7 ) << value << " = ";

   for ( c = 1; c <= 16; c++ ) {
      cout << ( value & displayMask ? '1' : '0' );
```

```
      value <<= 1;

      if ( c % 8 == 0 )
         cout << ' ';
   }

   cout << endl;
}
```

```
The result of left shifting
    960 = 00000011 11000000
8 bit positions using the left shift operator << is
  49152 = 11000000 00000000

The result of right shifting
    960 = 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000011
```

Fig. 16.11 Using the bitwise shift operators (part 2 of 2).

Bitwise assignment operators	
`&=`	Bitwise AND assignment operator.
`|=`	Bitwise inclusive OR assignment operator.
`^=`	Bitwise exclusive OR assignment operator.
`<<=`	Left shift assignment operator.
`>>=`	Right shift with sign extension assignment operator.

Fig. 16.12 The bitwise assignment operators.

Operators	Associativity	Type
`::` (unary; right to left) `::` (binary; left to right)	left to right	highest
`()` `[]` `.` `->`	left to right	highest
`++` `--` `+` `-` `!` `delete` `sizeof` `*` `&` `new`	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	shifting
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&`	left to right	bitwise AND
`^`	left to right	bitwise XOR
`|`	left to right	bitwise OR

Fig. 16.13 Operator precedence and associativity (part 1 of 2).

<table>
<tr><th>Operators</th><th>Associativity</th><th>Type</th></tr>
<tr><td>&&</td><td>left to right</td><td>logical AND</td></tr>
<tr><td>||</td><td>left to right</td><td>logical OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= %= &= |= ^= <<= >>=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>,</td><td>left to right</td><td>comma</td></tr>
</table>

Fig. 16.13 Operator precedence and associativity (part 2 of 2).

```
// Fig. 16.14: fig16_14.cpp
// Example using a bit field
#include <iostream.h>
#include <iomanip.h>

struct BitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};

void fillDeck( BitCard * );
void deal( BitCard * );

int main()
{
   BitCard deck[ 52 ];

   fillDeck( deck );
   deal( deck );
   return 0;
}

void fillDeck( BitCard *wDeck )
{
   for ( int i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = i % 13;
      wDeck[ i ].suit = i / 13;
      wDeck[ i ].color = i / 26;
   }
}

// Output cards in two column format. Cards 0-25 subscripted
// with k1 (column 1). Cards 26-51 subscripted k2 in (column 2.)
void deal( BitCard *wDeck )
{
   for ( int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      cout << "Card:" << setw( 3 ) << wDeck[ k1 ].face
           << "  Suit:" << setw( 2 ) << wDeck[ k1 ].suit
           << "  Color:" << setw( 2 ) << wDeck[ k1 ].color
           << "   " << "Card:" << setw( 3 ) << wDeck[ k2 ].face
           << "  Suit:" << setw( 2 ) << wDeck[ k2 ].suit
           << "  Color:" << setw( 2 ) << wDeck[ k2 ].color
```

```
            << endl;
    }
}
```

Fig. 16.14 Using bit fields to store a deck of cards.

```
Card:  0  Suit: 0  Color: 0  Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0  Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0  Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0  Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0  Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0  Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0  Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0  Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0  Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0  Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0  Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0  Card: 11  Suit: 2  Color: 1
Card: 12  Suit: 0  Color: 0  Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0  Card:  0  Suit: 3  Color: 1
Card:  1  Suit: 1  Color: 0  Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0  Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0  Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0  Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0  Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0  Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0  Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0  Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0  Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0  Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0  Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0  Card: 12  Suit: 3  Color: 1
```

Fig. 16.15 Output of the program in Fig. 16.14.

Prototype	Description
`int isdigit( int c )`	Returns **true** if **c** is a digit, and **false** otherwise.
`int isalpha( int c )`	Returns **true** if **c** is a letter, and **false** otherwise.
`int isalnum( int c )`	Returns **true** if **c** is a digit or a letter, and **false** otherwise.
`int isxdigit( int c )`	Returns **true** if **c** is a hexadecimal digit character, and **false** otherwise. (See Appendix C, "Number Systems," for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
`int islower( int c )`	Returns **true** if **c** is a lowercase letter, and **false** otherwise.
`int isupper( int c )`	Returns **true** if **c** is an uppercase letter; **false** otherwise.
`int tolower( int c )`	If **c** is an uppercase letter, **tolower** returns **c** as a lowercase letter. Otherwise, **tolower** returns the argument unchanged.

Fig. 16.16 Summary of the character handling library functions (part 1 of 2).

Prototype	Description
`int toupper( int c )`	If **c** is a lowercase letter, **toupper** returns **c** as an uppercase letter. Otherwise, **toupper** returns the argument unchanged.
`int isspace( int c )`	Returns **true** if **c** is a white-space character—newline (**'\n'**), space (**' '**), form feed (**'\f'**), carriage return (**'\r'**), horizontal tab (**'\t'**), or vertical tab (**'\v'**)—and **false** otherwise
`int iscntrl( int c )`	Returns **true** if **c** is a control character, and **false** otherwise.
`int ispunct( int c )`	Returns **true** if **c** is a printing character other than a space, a digit, or a letter, and **false** otherwise.
`int isprint( int c )`	Returns **true** value if **c** is a printing character including space (**' '**), and **false** otherwise.
`int isgraph( int c )`	Returns **true** if **c** is a printing character other than space (**' '**), and **false** otherwise.

Fig. 16.16 Summary of the character handling library functions (part 2 of 2).

```
// Fig. 16.17: fig16_17.cpp
// Using functions isdigit, isalpha, isalnum, and isxdigit
#include <iostream.h>
#include <ctype.h>

int main()
{
   cout << "According to isdigit:\n"
        << ( isdigit( '8' ) ? "8 is a" : "8 is not a" )
        << " digit\n"
        << ( isdigit( '#' ) ? "# is a" : "# is not a" )
        << " digit\n";
   cout << "\nAccording to isalpha:\n"
        << ( isalpha( 'A' ) ? "A is a" : "A is not a" )
        << " letter\n"
```

Fig. 16.17 Using **isdigit**, **isalpha**, **isalnum**, and **isxdigit** (part 1 of 2).

```
        << ( isalpha( 'b' ) ? "b is a" : "b is not a" )
        << " letter\n"
        << ( isalpha( '&' ) ? "& is a" : "& is not a" )
        << " letter\n"
        << ( isalpha( '4' ) ? "4 is a" : "4 is not a" )
        << " letter\n";
   cout << "\nAccording to isalnum:\n"
        << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
        << " digit or a letter\n"
        << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
        << " digit or a letter\n"
        << ( isalnum( '#' ) ? "# is a" : "# is not a" )
        << " digit or a letter\n";
   cout << "\nAccording to isxdigit:\n"
        << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
        << " hexadecimal digit\n"
        << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
        << " hexadecimal digit\n"
```

```
         << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
         << " hexadecimal digit\n"
         << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
         << " hexadecimal digit\n"
         << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
         << " hexadecimal digit" << endl;
   return 0;
}
```

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 16.17 Using **isdigit**, **isalpha**, **isalnum**, and **isxdigit** (part 2 of 2).

```
// Fig. 16.18: fig16_18.cpp
// Using functions islower, isupper, tolower, toupper
#include <iostream.h>
#include <ctype.h>

int main()
{
   cout << "According to islower:\n"
        << ( islower( 'p' ) ? "p is a" : "p is not a" )
        << " lowercase letter\n"
        << ( islower( 'P' ) ? "P is a" : "P is not a" )
        << " lowercase letter\n"
        << ( islower( '5' ) ? "5 is a" : "5 is not a" )
        << " lowercase letter\n"
        << ( islower( '!' ) ? "! is a" : "! is not a" )
        << " lowercase letter\n";
   cout << "\nAccording to isupper:\n"
        << ( isupper( 'D' ) ? "D is an" : "D is not an" )
        << " uppercase letter\n"
        << ( isupper( 'd' ) ? "d is an" : "d is not an" )
        << " uppercase letter\n"
        << ( isupper( '8' ) ? "8 is an" : "8 is not an" )
        << " uppercase letter\n"
        << ( isupper('$') ? "$ is an" : "$ is not an" )
        << " uppercase letter\n";
   cout << "\nu converted to uppercase is "
        << ( char ) toupper( 'u' )
        << "\n7 converted to uppercase is "
```

```
         << ( char ) toupper( '7' )
```

Fig. 16.18 Using **islower**, **isupper**, **tolower**, and **toupper** (part 1 of 2).

```
         << "\n$ converted to uppercase is "
         << ( char ) toupper( '$' )
         << "\nL converted to lowercase is "
         << ( char ) tolower( 'L' ) << endl;

    return 0;
}
```

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Fig. 16.18 Using **islower**, **isupper**, **tolower**, and **toupper** (part 2 of 2).

```
// Fig. 16.19: fig16_19.cpp
// Using functions isspace, iscntrl, ispunct, isprint, isgraph
#include <iostream.h>
#include <ctype.h>

int main()
{
```

Fig. 16.19 Using **isspace**, **iscntrl**, **ispunct**, **isprint**, and **isgraph** (part 1 of 3).

```
    cout << "According to isspace:\nNewline "
         << ( isspace( '\n' ) ? "is a" : "is not a" )
         << " whitespace character\nHorizontal tab "
         << ( isspace( '\t' ) ? "is a" : "is not a" )
         << " whitespace character\n"
         << ( isspace( '%' ) ? "% is a" : "% is not a" )
         << " whitespace character\n";
    cout << "\nAccording to iscntrl:\nNewline "
         << ( iscntrl( '\n' ) ? "is a" : "is not a" )
         << " control character\n"
         << ( iscntrl( '$' ) ? "$ is a" : "$ is not a" )
         << " control character\n";
    cout << "\nAccording to ispunct:\n"
         << ( ispunct( ';' ) ? "; is a" : "; is not a" )
         << " punctuation character\n"
         << ( ispunct( 'Y' ) ? "Y is a" : "Y is not a" )
         << " punctuation character\n"
         << ( ispunct('#') ? "# is a" : "# is not a" )
         << " punctuation character\n";
```

```
    cout << "\nAccording to isprint:\n"
         << ( isprint( '$' ) ? "$ is a" : "$ is not a" )
         << " printing character\nAlert "
         << ( isprint( '\a' ) ? "is a" : "is not a" )
         << " printing character\n";
    cout << "\nAccording to isgraph:\n"
         << ( isgraph( 'Q' ) ? "Q is a" : "Q is not a" )
         << " printing character other than a space\nSpace "
         << ( isgraph(' ') ? "is a" : "is not a" )
         << " printing character other than a space" << endl;

    return 0;
}
```

Fig. 16.19 Using **isspace**, **iscntrl**, **ispunct**, **isprint**, and **isgraph** (part 2 of 3).

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Fig. 16.19 Using **isspace**, **iscntrl**, **ispunct**, **isprint**, and **isgraph** (part 3 of 3).

Prototype	Description
`double atof( const char *nPtr )`	Converts the string **nPtr** to **double**.
`int atoi( const char *nPtr )`	Converts the string **nPtr** to **int**.
`long atol( const char *nPtr )`	Converts the string **nPtr** to long **int**.
`double strtod( const char *nPtr, char **endPtr )`	Converts the string **nPtr** to **double**.
`long strtol( const char *nPtr, char **endPtr, int base )`	Converts the string **nPtr** to **long**.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base )`	Converts the string **nPtr** to **unsigned long**.

Fig. 16.20 Summary of the string conversion functions of the general utilities library.

```
// Fig. 16.21: fig16_21.cpp
// Using atof
#include <iostream.h>
#include <stdlib.h>

int main()
{
   double d = atof( "99.0" );

   cout << "The string \"99.0\" converted to double is "
        << d << "\nThe converted value divided by 2 is "
        << d / 2.0 << endl;
   return 0;
}
```

```
The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5
```

Fig. 16.21 Using **atof**.

```
// Fig. 16.22: fig16_22.cpp
// Using atoi
#include <iostream.h>
#include <stdlib.h>

int main()
{
   int i = atoi( "2593" );

   cout << "The string \"2593\" converted to int is " << i
        << "\nThe converted value minus 593 is " << i - 593
        << endl;
   return 0;
}
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Fig. 16.22 Using **atoi**.

```
// Fig. 16.23: fig16_23.cpp
// Using atol
#include <iostream.h>
#include <stdlib.h>

int main()
{
   long l = atol( "1000000" );

   cout << "The string \"1000000\" converted to long is " << l
        << "\nThe converted value divided by 2 is " << l / 2
        << endl;
   return 0;
}
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Fig. 16.23 Using **atol**.

```
// Fig. 16.24: fig16_24.cpp
// Using strtod
#include <iostream.h>
#include <stdlib.h>

int main()
{
   double d;
   char *string = "51.2% are admitted", *stringPtr;
```

Fig. 16.24 Using **strtod** (part 1 of 2).

```

   d = strtod( string, &stringPtr );
   cout << "The string \"" << string
        << "\" is converted to the\ndouble value " << d
        << " and the string \"" << stringPtr << "\"" << endl;
   return 0;
}
```

```
The string "51.2% are admitted" is converted to the
double value 51.2 and the string "% are admitted"
```

Fig. 16.24 Using **strtod** (part 2 of 2).

```
// Fig. 16.25: fig16_25.cpp
// Using strtol
#include <iostream.h>
#include <stdlib.h>

int main()
{
   long x;
   char *string = "-1234567abc", *remainderPtr;
```

Fig. 16.25 Using **strtol** (part 1 of 2).

```

   x = strtol( string, &remainderPtr, 0 );
   cout << "The original string is \"" << string
        << "\"\nThe converted value is " << x
        << "\nThe remainder of the original string is \""
        << remainderPtr
        << "\"\nThe converted value plus 567 is "
        << x + 567 << endl;
   return 0;
}
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Fig. 16.25 Using **strtol** (part 2 of 2).

```
// Fig. 16.26: fig16_26.cpp
// Using strtoul
#include <iostream.h>
#include <stdlib.h>

int main()
{
   unsigned long x;
   char *string = "1234567abc", *remainderPtr;

   x = strtoul( string, &remainderPtr, 0 );
   cout << "The original string is \"" << string
        << "\"\nThe converted value is " << x
        << "\nThe remainder of the original string is \""
        << remainderPtr
        << "\"\nThe converted value minus 567 is "
        << x - 567 << endl;
   return 0;
}
```

Fig. 16.26 Using **strtoul** (part 1 of 2).

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Fig. 16.26 Using **strtoul** (part 2 of 2).

Prototype	Description
char *strchr(const char *s, int c)	
	Locates the first occurrence of character **c** in string **s**. If **c** is found, a pointer to **c** in **s** is returned. Otherwise, a **NULL** pointer is returned.
size_t strcspn(const char *s1, const char *s2)	
	Determines and returns the length of the initial segment of string **s1** consisting of characters not contained in string **s2**.
size_t strspn(const char *s1, const char *s2)	
	Determines and returns the length of the initial segment of string **s1** consisting only of characters contained in string **s2**.
char *strpbrk(const char *s1, const char *s2)	

Fig. 16.27 Search functions of the string handling library.

Prototype	Description
	Locates the first occurrence in string **s1** of any character in string **s2**. If a character from string **s2** is found, a pointer to the character in string **s1** is returned. Otherwise a **NULL** pointer is returned.
char *strrchr(const char *s, int c)	Locates the last occurrence of **c** in string **s**. If **c** is found, a pointer to **c** in string **s** is returned. Otherwise, a **NULL** pointer is returned.
char *strstr(const char *s1, const char *s2)	Locates the first occurrence in string **s1** of string **s2**. If the string is found, a pointer to the string in **s1** is returned. Otherwise, a **NULL** pointer is returned.

Fig. 16.27 Search functions of the string handling library.

```
// Fig. 16.28: fig16_28.cpp
// Using strchr
#include <iostream.h>
#include <string.h>

int main()
{
   char *string = "This is a test";
   char character1 = 'a', character2 = 'z';

   if ( strchr( string, character1 ) != NULL )
      cout << '\'' << character1 << "' was found in \""
           << string << "\".\n";
   else
      cout << '\'' << character1 << "' was not found in \""
           << string << "\".\n";

   if ( strchr( string, character2 ) != NULL )
      cout << '\'' << character2 << "' was found in \""
           << string << "\".\n";
   else
      cout << '\'' << character2 << "' was not found in \""
           << string << "\"." << endl;
   return 0;
}
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Fig. 16.28 Using **strchr**.

```
// Fig. 16.29: fig16_29.cpp
// Using strcspn
#include <iostream.h>
#include <string.h>

int main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "1234567890";
```

```

   cout << "string1 = " << string1 << "\nstring2 = " << string2
        << "\n\nThe length of the initial segment of string1"
        << "\ncontaining no characters from string2 = "
        << strcspn( string1, string2 ) << endl;
   return 0;
}
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

Fig. 16.29 Using **strcspn**.

```
// Fig. 16.30: fig16_30.cpp
// Using strpbrk
#include <iostream.h>
#include <string.h>

int main()
{
   char *string1 = "This is a test";
   char *string2 = "beware";

   cout << "Of the characters in \"" << string2 << "\"\n'"
        << *strpbrk( string1, string2 ) << '\''
        << " is the first character to appear in\n\""
        << string1 << '\"' << endl;
   return 0;
}
```

```
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
```

Fig. 16.30 Using **strpbrk**.

```
// Fig. 16.31: fig16_31.cpp
// Using strrchr
#include <iostream.h>
#include <string.h>

int main()
{
   char *string1 = "A zoo has many animals including zebras";
   int c = 'z';

   cout << "The remainder of string1 beginning with the\n"
        << "last occurrence of character '" << (char) c
        << "' is: \"" << strrchr( string1, c ) << '\"' << endl;
   return 0;
}
```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Fig. 16.31 Using **strrchr**.

```
// Fig. 16.32: fig16_32.cpp
// Using strspn
#include <iostream.h>
#include <string.h>

int main()
{
   char *string1 = "The value is 3.14159";
   char *string2 = "aehilsTuv ";

   cout << "string1 = " << string1
        << "\nstring2 = " << string2
        << "\n\nThe length of the initial segment of string1\n"
        << "containing only characters from string2 = "
        << strspn( string1, string2 ) << endl;
   return 0;
}
```

Fig. 16.32 Using **strspn** (part 1 of 2).

```
string1 = The value is 3.14159
string2 = aehilsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

Fig. 16.32 Using **strspn** (part 2 of 2).

```
// Fig. 16.33: fig16_33.cpp
// Using strstr
#include <iostream.h>
#include <string.h>

int main()
{
   char *string1 = "abcdefabcdef";
   char *string2 = "def";

   cout << "string1 = " << string1 << "\nstring2 = " << string2
        << "\n\nThe remainder of string1 beginning with the\n"
        << "first occurrence of string2 is: "
        << strstr( string1, string2 ) << endl;
   return 0;
}
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Fig. 16.33 Using **strstr**.

Prototype	Description
void *memcpy(void *s1, const void *s2, size_t n)	Copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. A pointer to the resulting object is returned.
void *memmove(void *s1, const void *s2, size_t n)	Copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The copy is performed as if the characters are first copied from the object pointed to by **s2** into a temporary array, then from the temporary array into the object pointed to by **s1**. A pointer to the resulting object is returned.
int memcmp(const void *s1, const void *s2, size_t n)	Compares the first **n** characters of the objects pointed to by **s1** and **s2**. The function returns **0**, less than **0**, or greater than **0** if **s1** is equal to, less than, or greater than **s2**.
void *memchr(const void *s, int c, size_t n)	Locates the first occurrence of **c** (converted to **unsigned char**) in the first **n** characters of the object pointed to by **s**. If **c** is found, a pointer to **c** in the object is returned. Otherwise, **0** is returned.
void *memset(void *s, int c, size_t n)	Copies **c** (converted to **unsigned char**) into the first **n** characters of the object pointed to by **s**. A pointer to the result is returned.

Fig. 16.34 The memory functions of the string handling library.

```
// Fig. 16.35: fig16_35.cpp
// Using memcpy
#include <iostream.h>
#include <string.h>

int main()
{
   char s1[ 17 ], s2[]  = "Copy this string";

   memcpy( s1, s2, 17 );
   cout << "After s2 is copied into s1 with memcpy,\n"
        << "s1 contains \"" << s1 << '\"' << endl;
   return 0;
}
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Fig. 16.35 Using **memcpy**.

```cpp
// Fig. 16.36: fig16_36.cpp
// Using memmove
#include <iostream.h>
#include <string.h>

int main()
{
   char x[] = "Home Sweet Home";

   cout << "The string in array x before memmove is: " << x;
   cout << "\nThe string in array x after memmove is:  "
        << (char *) memmove( x, &x[ 5 ], 10 ) << endl;
   return 0;
}
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is:  Sweet Home Home
```

Fig. 16.36 Using **memmove**.

```cpp
// Fig. 16.37: fig16_37.cpp
// Using memcmp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

int main()
{
   char s1[] = "ABCDEFG", s2[] = "ABCDXYZ";

   cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
        << "\nmemcmp(s1, s2, 4) = " << setw( 3 )
        << memcmp( s1, s2, 4 ) << "\nmemcmp(s1, s2, 7) = "
        << setw( 3 ) << memcmp( s1, s2, 7 )
        << "\nmemcmp(s2, s1, 7) = " << setw( 3 )
        << memcmp( s2, s1, 7 ) << endl;
   return 0;
}
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =   0
memcmp(s1, s2, 7) = -19
memcmp(s2, s1, 7) =  19
```

Fig. 16.37 Using **memcmp**.

```
// Fig. 16.38: fig16_38.cpp
// Using memchr
#include <iostream.h>
#include <string.h>

int main()
{
   char *s = "This is a string";
```

Fig. 16.38 Using **memchr** (part 1 of 2).

```

   cout << "The remainder of s after character 'r' "
        << "is found is \"" << (char *) memchr( s, 'r', 16 )
        << '\"' << endl;
   return 0;
}
```

```
The remainder of s after character 'r' is found is "ring"
```

Fig. 16.38 Using **memchr** (part 2 of 2).

```
// Fig. 16.39: fig16_39.cpp
// Using memset
#include <iostream.h>
#include <string.h>

int main()
{
   char string1[ 15 ] = "BBBBBBBBBBBBBB";

   cout << "string1 = " << string1 << endl;
   cout << "string1 after memset = "
        << (char *) memset( string1, 'b', 7 ) << endl;
   return 0;
}
```

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

Fig. 16.39 Using **memset**.

Prototype	Description
`char *strerror( int errornum )`	
	Maps **errornum** into a full text string in a system dependent manner. A pointer to the string is returned.

Fig. 16.40 Another string manipulation function of the string handling library.

```
// Fig. 16.41: fig16_41.cpp
// Using strerror
#include <iostream.h>
#include <string.h>

int main()
{
   cout << strerror( 2 ) << endl;
   return 0;
}
```

```
No such file or directory
```

Fig. 16.41 Using `strerror`.

Illustrations List *(Main Page)*

Fig. 17.1 The predefined symbolic constants.

Symbolic constant	Description
`__LINE__`	The line number of the current source code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form `"Mmm dd yyyy"` such as `"Jan 19 1994"`).
`__TIME__`	The time the source file is compiled (a string literal of the form `"hh:mm:ss"`).
`__STDC__`	The integer constant 1. This is intended to indicate that the implementation is ANSI compliant.

Fig. 17.1 The predefined symbolic constants.

Illustrations List *(Main Page)*

Identifier	Description
va_list	A type suitable for holding information needed by macros **va_start**, **va_arg**, and **va_end**. To access the arguments in a variable-length argument list, an object of type **va_list** must be declared.
va_start	A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with **va_list** for use by the **va_arg** and **va_end** macros.
va_arg	A macro that expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of **va_arg** modifies the object declared with **va_list** so that the object points to the next argument in the list.
va_end	A macro that facilitates a normal return from a function whose variable-length argument list was referred to by the **va_start** macro.

Fig. 18.1 The type and the macros defined in header **stdarg.h** .

```
// Fig. 18.2: fig18_02.cpp
// Using variable-length argument lists
#include <iostream.h>
#include <iomanip.h>
#include <stdarg.h>

double average( int, ... );

int main()
{
   double w = 37.5, x = 22.5, y = 1.7, z = 10.2;

   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 1 ) << "w = " << w << "\nx = " << x
        << "\ny = " << y << "\nz = " << z << endl;
   cout << setprecision( 3 ) << "\nThe average of w and x is "
        << average( 2, w, x )
        << "\nThe average of w, x, and y is "
        << average( 3, w, x, y )
        << "\nThe average of w, x, y, and z is "
        << average( 4, w, x, y, z ) << endl;
   return 0;
}

double average( int i, ... )
{
   double total = 0;
```

Fig. 18.2 Using variable-length argument lists (part 1 of 2).

```
   va_list ap;

   va_start( ap, i );

```

```
   for ( int j = 1; j <= i; j++ )
      total += va_arg( ap, double );

   va_end( ap );

   return total / i;
}
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Fig. 18.2 Using variable-length argument lists (part 2 of 2).

```
// Fig. 18.3: fig18_03.cpp
// Using command-line arguments
#include <iostream.h>
#include <fstream.h>

int main( int argc, char *argv[] )
{
   if ( argc != 3 )
      cout << "Usage: copy infile outfile" << endl;
   else {
      ifstream inFile( argv[ 1 ], ios::in );
      if ( !inFile )
         cout << argv[ 1 ] << " could not be opened" << endl;

      ofstream outFile( argv[ 2 ], ios::out );
      if ( !outFile )
         cout << argv[ 2 ] << " could not be opened" << endl;

      while ( !inFile.eof() )
         outFile.put( static_cast< char >( inFile.get() ) );
   }

   return 0;
}
```

Fig. 18.3 Using command-line arguments.

```
// Fig. 18.4: fig18_04.cpp
// Using the exit and atexit functions
#include <iostream.h>
#include <stdlib.h>

void print( void );

int main()
{
   atexit( print );       // register function print
   cout << "Enter 1 to terminate program with function exit"
        << "\nEnter 2 to terminate program normally\n";
```

```

    int answer;
    cin >> answer;

    if ( answer == 1 ) {
       cout << "\nTerminating program with function exit\n";
       exit( EXIT_SUCCESS );
    }

    cout << "\nTerminating program by reaching the of main"
         << endl;

    return 0;
 }

 void print( void )
 {
    cout << "Executing function print at program termination\n"
         << "Program terminated" << endl;
 }
```

Fig. 18.4 Using functions **exit** and **atexit** (part 1 of 2).

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
: 2

Terminating program by reaching end of main
Executing function print at program termination
Program terminated
```

Fig. 18.4 Using functions **exit** and **atexit** (part 2 of 2).

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to **abort**).
SIGFPE	An erroneous arithmetic operation, such as a divide-by-zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SIGINT	Receipt of an interactive attention signal.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.

Fig. 18.5 The signals defined in header **signal.h**.

```
// Fig. 18.6: fig18_06.cpp
// Using signal handling
#include <iostream.h>
#include <iomanip.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signal_handler( int );

int main()
{
   signal( SIGINT, signal_handler );
   srand( time( 0 ) );

   for ( int i = 1; i < 101; i++ ) {
      int x = 1 + rand() % 50;

      if ( x == 25 )
         raise( SIGINT );

      cout << setw( 4 ) << i;

      if ( i % 10 == 0 )
         cout << endl;
   }

   return 0;
}

void signal_handler( int signalValue )
{
   cout << "\nInterrupt signal (" << signalValue
        << ") received.\n"
        << "Do you wish to continue (1 = yes or 2 = no)? ";

   int response;
   cin >> response;

   while ( response != 1 && response != 2 ) {
      cout << "(1 = yes or 2 = no)? ";
      cin >> response;
   }

   if ( response == 1 )
      signal( SIGINT, signal_handler );
   else
      exit( EXIT_SUCCESS );
}
```

Fig. 18.6 Using signal handling (part 1 of 2).

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29  30
 31  32  33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60
 61  62  63  64  65  66  67  68  69  70
 71  72  73  74  75  76  77  78  79  80
 81  82  83  84  85  86  87  88
Interrupt signal (4) received.
Do you wish to continue (1 = yes or 2 = no)? 1
 89  90
 91  92  93  94  95  96  97  98  99 100
```

Fig. 18.6 Using signal handling (part 2 of 2).

```
// Fig. 18.7: fig18_07.cpp
// Using goto
#include <iostream.h>

int main()
{
   int count = 1;

   start:                        // label
      if ( count > 10 )
         goto end;

      cout << count << "  ";
      ++count;
      goto start;

   end:                          // label
      cout << endl;

   return 0;
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 18.7 Using `goto`.

```
// Fig. 18.8: fig18_08.cpp
// An example of a union
#include <iostream.h>

union Number {
   int x;
   float y;
};
```

Fig. 18.8 Printing the value of a `union` in both member data types (part 1 of 2).

```
int main()
{
   Number value;

   value.x = 100;
   cout << "Put a value in the integer member\n"
        << "and print both members.\nint:   "
        << value.x << "\nfloat: " << value.y << "\n\n";

   value.y = 100.0;
   cout << "Put a value in the floating member\n"
        << "and print both members.\nint:   "
        << value.x << "\nfloat: " << value.y << endl;
   return 0;
}
```

```
Put a value in the integer member
and print both members.
int:   100
float: 3.504168e-16

Put a value in the floating member
and print both members.
int:   0
float: 100
```

Fig. 18.8 Printing the value of a **union** in both member data types (part 2 of 2).

```
// Fig. 18.9: fig18_09.cpp
// Using an anonymous union
#include <iostream.h>

int main()
{
   // Declare an anonymous union.
```

Fig. 18.9 Using an anonymous **union** (part 1 of 2).

```
   // Note that members b, d, and f share the same space.
   union {
      int b;
      double d;
      char *f;
   };

   // Declare conventional local variables
   int a = 1;
   double c = 3.3;
   char *e = "Anonymous";

   // Assign a value to each union member
   // successively and print each.
   cout << a << ' ';
   b = 2;
   cout << b << endl;

   cout << c << ' ';
   d = 4.4;
   cout << d << endl;
```

```

   cout << e << ' ';
   f = "union";
   cout << f << endl;

   return 0;
}
```

```
1 2
3.3 4.4
Anonymous union
```

Fig. 18.9 Using an anonymous **`union`** (part 2 of 2).

Illustrations List *(Main Page)*

```
// Fig. 19.1: fig19_01.cpp
// Demonstrating string assignment and concatenation
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s1( "cat" ), s2, s3;

   s2 = s1;           // assign s1 to s2 with =
   s3.assign( s1 );  // assign s1 to s3 with assign()
   cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: "
        << s3 << "\n\n";

   // modify s2 and s3
   s2[ 0 ] = s3[ 2 ] = 'r';

   cout << "After modification of s2 and s3:\n"
        << "s1: " << s1 << "\ns2: " << s2 << "\ns3: ";

   // demonstrating member function at()
   int len = s3.length();
   for ( int x = 0; x < len; ++x )
      cout << s3.at( x );

   // concatenation
   string s4( s1 + "apult" ), s5;  // declare s4 and s5

   // overloaded +=
   s3 += "pet";             // create "carpet"
   s1.append( "acomb" );    // create "catacomb"

   // append subscript locations 4 thru the end of s1 to
   // create the string "comb" (s5 was initially empty)
   s5.append( s1, 4, s1.size() );

   cout << "\n\nAfter concatenation:\n" << "s1: " << s1
        << "\ns2: " << s2 << "\ns3: " << s3 << "\ns4: " << s4
        << "\ns5: " << s5 << endl;

   return 0;
}
```

Fig. 19.1 Demonstrating **string** assignment and concatenation (part 1 of 2).

```
s1: cat
s2: cat
s3: cat

After modification of s2 and s3:
s1: cat
s2: rat
s3: car

After concatenation:
s1: catacomb
s2: rat
s3: carpet
s4: catapult
s5: comb
```

Fig. 19.1 Demonstrating **string** assignment and concatenation (part 2 of 2).

```cpp
// Fig. 19.2: fig19_02.cpp
// Demonstrating string comparison capabilities
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s1( "Testing the comparison functions." ),
          s2("Hello" ), s3( "stinger" ), z1( s2 );

   cout << "s1: " << s1 << "\ns2: " << s2
        << "\ns3: " << s3 << "\nz1: " << z1 << "\n\n";
```

Fig. 19.2 Comparing **string**s (part 1 of 3).

```cpp

   // comparing s1 and z1
   if ( s1 == z1 )
      cout << "s1 == z1\n";
   else { // s1 != z1
      if ( s1 > z1 )
         cout << "s1 > z1\n";
      else // s1 < z1
         cout << "s1 < z1\n";
   }

   // comparing s1 and s2
   int f = s1.compare( s2 );

   if ( f == 0)
      cout << "s1.compare( s2 ) == 0\n";
   else if ( f > 0 )
      cout << "s1.compare( s2 ) > 0\n";
   else  // f < 0
      cout << "s1.compare( s2 ) < 0\n";

   // comparing s1 (elements 2 - 5) and s3 (elements 0 - 5)
   f = s1.compare( 2, 5, s3, 0, 5 );

   if ( f == 0 )
```

```
      cout << "s1.compare( 2, 5, s3, 0, 5 ) == 0\n";
   else if ( f > 0 )
      cout << "s1.compare( 2, 5, s3, 0, 5 ) > 0\n";
   else  // f < 0
      cout << "s1.compare( 2, 5, s3, 0, 5 ) < 0\n";

   // comparing s2 and z1
   f = z1.compare( 0, s2.size(), s2 );

   if ( f == 0 )
      cout << "z1.compare( 0, s2.size(), s2 ) == 0" << endl;
   else if ( f > 0 )
      cout << "z1.compare( 0, s2.size(), s2 ) > 0" << endl;
   else  // f < 0
      cout << "z1.compare( 0, s2.size(), s2 ) < 0" << endl;

   return 0;
}
```

Fig. 19.2 Comparing **string**s (part 2 of 3).

```
s1: Testing the comparison functions.
s2: Hello
s3: stinger
z1: Hello

s1 > z1
s1.compare( s2 ) > 0
s1.compare( 2, 5, s3, 0, 5 ) == 0
z1.compare( 0, s2.size(), s2 ) == 0
```

Fig. 19.2 Comparing **string**s (part 3 of 3).

```
// Fig. 19.3: fig19_03.cpp
// Demonstrating function substr
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s( "The airplane flew away." );

   // retrieve the substring "plane" which
   // begins at subscript 7 and consists of 5 elements
   cout << s.substr( 7, 5 ) << endl;

   return 0;
}
```

```
plane
```

Fig. 19.3 Using function **substr** to extract a substring from a **string**.

```
// Fig. 19.4: fig19_04.cpp
// Using the swap function to swap two strings
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string first( "one" ), second( "two" );

   cout << "Before swap:\n first: " << first
        << "\nsecond: " << second;
   first.swap( second );
   cout << "\n\nAfter swap:\n first: " << first
        << "\nsecond: " << second << endl;

   return 0;
}
```

```
Before swap:
 first: one
second: two

After swap:
 first: two
second: one
```

Fig. 19.4 Using function **swap** to swap two **string**s.

```
// Fig. 19.5: fig19_05.cpp
// Demonstrating functions related to size and capacity
#include <iostream>
#include <string>
using namespace std;

void printStats( const string & );
```

Fig. 19.5 Printing **string** characteristics (part 1 of 3).

```

int main()
{
   string s;

   cout << "Stats before input:\n";
   printStats( s );

   cout << "\n\nEnter a string: ";
   cin >> s;  // delimited by whitespace
   cout << "The string entered was: " << s;

   cout << "\nStats after input:\n";
   printStats( s );

   s.resize( s.length() + 10 );
   cout << "\n\nStats after resizing by (length + 10):\n";
   printStats( s );

   cout << endl;
   return 0;
```

```
}

void printStats( const string &str )
{
   cout << "capacity: " << str.capacity()
        << "\nmax size: " << str.max_size()
        << "\nsize: " << str.size()
        << "\nlength: " << str.length()
        << "\nempty: " << ( str.empty() ? "true": "false" );
}
```

Fig. 19.5 Printing **string** characteristics (part 2 of 3).

```
Stats before input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Stats after input:
capacity: 31
max size: 4294967293
size: 6
length: 6
empty: false

Stats after resizing by (length + 10):
capacity: 31
max size: 4294967293
size: 16
length: 16
empty: false
```

Fig. 19.5 Printing **string** characteristics (part 3 of 3).

```
// Fig. 19.6: fig19_06.cpp
// Demonstrating the string find functions
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // compiler concatenates all parts into one string literal
   string s( "The values in any left subtree"
             "\nare less than the value in the"
             "\nparent node and the values in"
             "\nany right subtree are greater"
             "\nthan the value in the parent node" );
```

Fig. 19.6 Program that demonstrates the **string find** functions (part 1 of 2).

```

   // find "subtree" at locations 23 and 102
   cout << "Original string:\n" << s
```

```
        << "\n\n(find) \"subtree\" was found at: "
        << s.find( "subtree" )
        << "\n(rfind) \"subtree\" was found at: "
        << s.rfind( "subtree" );

   // find 'p' in parent at locations 62 and 144
   cout << "\n(find_first_of) character from \"qpxz\" at: "
        << s.find_first_of( "qpxz" )
        << "\n(find_last_of) character from \"qpxz\" at: "
        << s.find_last_of( "qpxz" );

   // find 'b' at location 25
   cout << "\n(find_first_not_of) first character not\n"
        << "   contained in \"heTv lusinodrpayft\": "
        << s.find_first_not_of( "heTv lusinodrpayft" );

   // find '\n' at location 121
   cout << "\n(find_last_not_of) first character not\n"
        << "   contained in \"heTv lusinodrpayft\": "
        << s.find_last_not_of( "heTv lusinodrpayft" ) << endl;

   return 0;
}
```

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

(find) "subtree" was found at: 23
(rfind) "subtree" was found at: 102
(find_first_of) character from "qpxz" at: 62
(find_last_of) character from "qpxz" at: 144
(find_first_not_of) first character not
   contained in "heTv lusinodrpayft": 25
(find_last_not_of) first character not
   contained in "heTv lusinodrpayft": 121
```

Fig. 19.6 Program that demonstrates the `string find` functions (part 2 of 2).

```
// Fig. 19.7: fig19_07.cpp
// Demonstrating functions erase and replace
#include <iostream>
#include <string>
using namespace std;

int main()
{
   // compiler concatenates all parts into one string
   string s( "The values in any left subtree"
             "\nare less than the value in the"
             "\nparent node and the values in"
             "\nany right subtree are greater"
             "\nthan the value in the parent node" );

   // remove all characters from location 62
   // through the end of s
   s.erase( 62 );

   // output the new string
   cout << "Original string after erase:\n" << s
```

```
          << "\n\nAfter first replacement:\n";

   // replace all spaces with a period
   int x = s.find( " " );
   while ( x < string::npos ) {
      s.replace( x, 1, "." );
      x = s.find( " ", x + 1 );
   }

   cout << s << "\n\nAfter second replacement:\n";

   // replace all periods with two semicolons
   // NOTE: this will overwrite characters
   x = s.find( "." );
   while ( x < string::npos ) {
      s.replace( x, 2, "xxxxx;;yyy", 5, 2 );
      x = s.find( ".", x + 1 );
   }

   cout << s << endl;
   return 0;
}
```

Fig. 19.7 Demonstrating functions **erase** and **replace** (part 1 of 2).

```
Original string after erase:
The values in any left subtree
are less than the value in the


After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the


After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

Fig. 19.7 Demonstrating functions **erase** and **replace** (part 2 of 2).

```
// Fig. 19.8: fig19_08.cpp
// Demonstrating the string insert functions.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s1( "beginning end" ),
          s2( "middle " ), s3( "12345678" ), s4( "xx" );

   cout << "Initial strings:\ns1: " << s1
        << "\ns2: " << s2 << "\ns3: " << s3
        << "\ns4: " << s4 << "\n\n";

   // insert "middle" at location 10
   s1.insert( 10, s2 );

   // insert "xx" at location 3 in s3
   s3.insert( 3, s4, 0, string::npos );
```

```

   cout << "Strings after insert:\ns1: " << s1
        << "\ns2: " << s2 << "\ns3: " << s3
        << "\ns4: " << s4 << endl;

   return 0;
}
```

```
Initial strings:
s1: beginning end
s2: middle
s3: 12345678
s4: xx

Strings after insert:
s1: beginning middle end
s2: middle
s3: 123xx45678
s4: xx
```

Fig. 19.8 Demonstrating the **string insert** functions.

```
// Fig. 19.9: fig19_09.cpp
// Converting to C-style strings.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s( "STRINGS" );
   const char *ptr1 = 0;
   int len = s.length();
   char *ptr2 = new char[ len + 1 ]; // including null

   // Assign to pointer ptr1 the const char * returned by
   // function data(). NOTE: this is a potentially dangerous
   // assignment. If the string is modified, the pointer
   // ptr1 can become invalid.
   ptr1 = s.data();

   // copy characters out of string into allocated memory
   s.copy( ptr2, len, 0 );
   ptr2[ len ] = 0;  // add null terminator

   // output
   cout << "string s is " << s
        << "\ns converted to a C-Style string is "
        << s.c_str() << "\nptr1 is ";

   for ( int k = 0; k < len; ++k )
      cout << *( ptr1 + k );   // use pointer arithmetic

   cout << "\nptr2 is " << ptr2 << endl;
   delete [] ptr2;
   return 0;
}
```

Fig. 19.9 Converting **string**s to C-style strings and character arrays (part 1 of 2).

```
string s is STRINGS
s converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

Fig. 19.9 Converting **strings** to C-style strings and character arrays (part 2 of 2).

```
// Fig. 19.10: fig19_10.cpp
// Using an iterator to output a string.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s( "Testing iterators" );
   string::const_iterator i1 = s.begin();

   cout << "s = " << s
        << "\n(Using iterator i1) s is: ";

   while ( i1 != s.end() ) {
      cout << *i1;     // dereference iterator to get char
      ++i1;            // advance iterator to next char
   }

   cout << endl;
   return 0;
}
```

```
s = Testing iterators
(Using iterator i1) s is: Testing iterators
```

Fig. 19.10 Using an iterator to output a **string**.

```
// Fig. 19.11: fig19_11.cpp
// Using a dynamically allocated ostringstream object.
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

main()
{
   ostringstream outputString;
   string s1( "Output of several data types " ),
          s2( "to an ostringstream object:" ),
          s3( "\n        double: " ),
          s4( "\n           int: " ),
          s5( "\naddress of int: " );
   double d = 123.4567;
   int i = 22;

   outputString << s1 << s2 << s3 << d << s4 << i << s5 << &i;
   cout << "outputString contains:\n" << outputString.str();

```

```
   outputString << "\nmore characters added";
   cout << "\n\nafter additional stream insertions,\n"
        << "outputString contains:\n" << outputString.str()
        << endl;

   return 0;
}
```

```
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0068FD0C

after additional stream insertions,
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0068FD0C
more characters added
```

Fig. 19.11 Using a dynamically allocated `ostringstream` object.

```
// Fig. 19.12: fig19_12.cpp
// Demonstrating input from an istringstream object.
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

```

Fig. 19.12 Demonstrating input from an `istringstream` object (part 1 of 2).

```
main()
{
   string input( "Input test 123 4.7 A" );
   istringstream inputString( input );
   string string1, string2;
   int i;
   double d;
   char c;

   inputString >> string1 >> string2 >> i >> d >> c;

   cout << "The following inputs were extracted\n"
        << "from the istringstream object:"
        << "\nstring: " << string1
        << "\nstring: " << string2
        << "\n   int: " << i
        << "\ndouble: " << d
        << "\n  char: " << c;

   // attempt to read from empty stream
   long l;

   if ( inputString >> l )
      cout << "\n\nlong value is: " << l << endl;
   else
```

```
      cout << "\n\ninputString is empty" << endl;

   return 0;
}
```

```
The following items were extracted
from the istringstream object:
String: Input
String: test
   int: 123
double: 4.7
  char: A

inputString is empty
```

Fig. 19.12 Demonstrating input from an **`istringstream`** object (part 2 of 2).

Illustrations List *(Main Page)*

Standard Library container class	Description
Sequence Containers	
vector	rapid insertions and deletions at back direct access to any element
deque	rapid insertions and deletions at front or back direct access to any element
list	doubly-linked list, rapid insertion and deletion anywhere
Associative Containers	
set	rapid lookup, no duplicates allowed
multiset	rapid lookup, duplicates allowed
map	one-to-one mapping, no duplicates allowed, rapid key-based lookup
multimap	one-to-many mapping, duplicates allowed, rapid key-based lookup
Container Adapters	
stack	last-in-first-out (LIFO)
queue	first-in-first-out (FIFO)
priority_queue	highest priority element is always the first element out

Fig. 20.1 Standard Library container classes.

Common member functions for all STL containers	Description
default constructor	A constructor to provide a default initialization of the container. Normally, each container has several constructors that provide a variety of initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.
empty	Returns **true** if there are no elements in the container; otherwise, returns **false**.
max_size	Returns the maximum number of elements for a container.
size	Returns the number of elements currently in the container.
operator=	Assigns one container to another.
operator<	Returns **true** if the first container is less than the second container; otherwise, returns **false**.
operator<=	Returns **true** if the first container is less than or equal to the second container; otherwise, returns **false**.
operator>	Returns **true** the first container is greater than the second container; otherwise, returns **false**.

Fig. 20.2 Common functions for all STL containers.

Common member functions for all STL containers	Description
operator>=	Returns **true** if the first container is greater than or equal to the second container; otherwise, returns **false**.
operator==	Returns **true** if the first container is equal to the second container; otherwise, returns **false**.
operator!=	Returns **true** if the first container is not equal to the second container; otherwise, returns **false**.
swap	Swaps the elements of two containers.
Functions that are only found in first-class containers	
begin	The two versions of this function return either an **iterator** or a **const_iterator** that refers to the first element of the container.
end	The two versions of this function return either an **iterator** or a **const_iterator** that refers to the next position after the end of the container.
rbegin	The two versions of this function return either a **reverse_iterator** or a **const_reverse_iterator** that refers to the last element of the container.
rend	The two versions of this function return either a **reverse_iterator** or a **const_reverse_iterator** that refers to the position before the first element of the container.
erase	Erases one or more elements from the container.
clear	Erases all elements from the container.

Fig. 20.2 Common functions for all STL containers.

.

Standard Library container header files	
<vector>	
<list>	
<deque>	
<queue>	contains both **queue** and **priority_queue**
<stack>	
<map>	contains both **map** and **multimap**
<set>	contains both **set** and **multiset**
<bitset>	

Fig. 20.3 Standard Library container header files.

typedef	Description
value_type	The type of element stored in the container.
reference	A reference to the type of element stored in the container.
const_reference	A constant reference to the type of element stored in the container. Such a reference can only be used for *reading* elements in the container and for performing **const** operations.
pointer	A pointer to the type of element stored in the container.
iterator	An iterator that points to the type of element stored in the container.
const_iterator	A constant iterator that points to the type of element stored in the container and can only be used to *read* elements.
reverse_iterator	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.
const_reverse_iterator	A constant reverse iterator to the type of element stored in the container and can only be used to *read* elements. This type of iterator is for iterating through a container in reverse.
difference_type	The type of the result of subtracting two iterators that refer to the same container (**operator-** is not defined for iterators of **list**s and associative containers).
size_type	The type used to count items in a container and index through a sequence container (cannot index through a **list**).

Fig. 20.4 Common **typedef**s found in first-class containers.

```
// Fig. 20.5: fig20_05.cpp
// Demonstrating input and output with iterators.
#include <iostream>
#include <iterator>

using namespace std;

int main()
{
   cout << "Enter two integers: ";

```

Fig. 20.5 Demonstrating input and output stream iterators (part 1 of 2).

```
   istream_iterator< int > inputInt( cin );
   int number1, number2;

   number1 = *inputInt;  // read first int from standard input
   ++inputInt;           // move iterator to next input value
   number2 = *inputInt;  // read next int from standard input

   cout << "The sum is: ";

   ostream_iterator< int > outputInt( cout );

   *outputInt = number1 + number2;  // output result to cout
```

```
    cout << endl;
    return 0;
}
```

```
Enter two integers: 12 25
The sum is: 37
```

Fig. 20.5 Demonstrating input and output stream iterators (part 2 of 2).

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end of the container) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retain their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning of the container). Forward iterators support multi-pass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Fig. 20.6 Iterator categories.

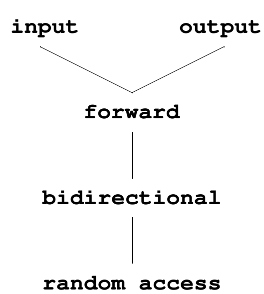


Fig. 20.7 Iterator category hierarchy.

Container	Type of iterator supported
Sequence containers	
vector	random access
deque	random access
list	bidirectional
Associative containers	
set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional
Container adapters	
stack	no iterators supported
queue	no iterators supported
priority_queue	no iterators supported

Fig. 20.8 Iterator types supported by each Standard Library container.

Predefined **typedef**s for iterator types	Direction of ++	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
const_reverse_iterator	backward	read

Fig. 20.9 Predefined iterator **typedef**s.

Iterator operation	Description
All iterators	
++p	preincrement an iterator
p++	postincrement an iterator
Input iterators	
***p**	dereference an iterator for use as an *rvalue*
p = p1	assign one iterator to another
p == p1	compare iterators for equality
p != p1	compare iterators for inequality
Output iterators	
***p**	dereference an iterator (for use as an *lvalue*)

Fig. 20.10 Iterator operations for each type of iterator.

Iterator operation	Description
p = p1	assign one iterator to another
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
--p	predecrement an iterator
p--	postdecrement an iterator
Random-access iterators	
p += i	Increment the iterator **p** by **i** positions.
p -= i	Decrement the iterator **p** by **i** positions.
p + i	Results in an iterator positioned at **p** incremented by **i** positions.
p - i	Results in an iterator positioned at **p** decremented by **i** positions.
p[i]	Return a reference to the element offset from **p** by **i** positions
p < p1	Return **true** if iterator **p** is less than iterator **p1** (i.e., iterator **p** is before iterator **p1** in the container); otherwise, return **false**.
p <= p1	Return **true** if iterator **p** is less than or equal to iterator **p1** (i.e., iterator **p** is before iterator **p1** or at the same location as iterator **p1** in the container); otherwise, return **false**.
p > p1	Return **true** if iterator **p** is greater than iterator **p1** (i.e., iterator **p** is after iterator **p1** in the container); otherwise, return **false**.
p >= p1	Return **true** if iterator **p** is greater than or equal to iterator **p1** (i.e., iterator **p** is after iterator **p1** or at the same location as iterator **p1** in the container); otherwise, return **false**.

Fig. 20.10 Iterator operations for each type of iterator.

Mutating-sequence algorithms		
copy()	**remove()**	**reverse_copy()**
copy_backward()	**remove_copy()**	**rotate()**
fill()	**remove_copy_if()**	**rotate_copy()**
fill_n()	**remove_if()**	**stable_partition()**
generate()	**replace()**	**swap()**
generate_n()	**replace_copy()**	**swap_ranges()**
iter_swap()	**replace_copy_if()**	**transform()**
partition()	**replace_if()**	**unique()**
random_shuffle()	**reverse()**	**unique_copy()**

Fig. 20.11 Mutating-sequence algorithms.

Non-mutating sequence algorithms		
`adjacent-find()`	`equal()`	`mismatch()`
`count()`	`find()`	`search()`
`count_if()`	`for_each()`	`search_n()`

Fig. 20.12 Non-mutating sequence algorithms.

Numerical algorithms from header file `<numeric>`
`accumulate()`
`inner_product()`
`partial_sum()`
`adjacent_difference()`

Fig. 20.13 Numerical algorithms from header file `<numeric>`.

```
// Fig. 20.14: fig20_14.cpp
// Testing Standard Library vector class template
#include <iostream>
#include <vector>

using namespace std;

template < class T >
void printVector( const vector< T > &vec );

int main()
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > v;

   cout << "The initial size of v is: " << v.size()
        << "\nThe initial capacity of v is: " << v.capacity();
   v.push_back( 2 );  // method push_back() is in
   v.push_back( 3 );  // every sequence container
   v.push_back( 4 );
   cout << "\nThe size of v is: " << v.size()
        << "\nThe capacity of v is: " << v.capacity();
   cout << "\n\nContents of array a using pointer notation: ";

   for ( int *ptr = a; ptr != a + SIZE; ++ptr )
      cout << *ptr << ' ';

   cout << "\nContents of vector v using iterator notation: ";
   printVector( v );

   cout << "\nReversed contents of vector v: ";

   vector< int >::reverse_iterator p2;

   for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
      cout << *p2 << ' ';
```

```
   cout << endl;
   return 0;
}
```

Fig. 20.14 Demonstrating Standard Library **vector** class template (part 1 of 2).

```

template < class T >
void printVector( const vector< T > &vec )
{
   vector< T >::const_iterator p1;

   for ( p1 = vec.begin(); p1 != vec.end(); ++p1 )
      cout << *p1 << ' ';
}
```

```
The initial size of v is: 0
The initial capacity of v is: 0
The size of v is: 3
The capacity of v is: 4

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2
```

Fig. 20.14 Demonstrating Standard Library **vector** class template (part 2 of 2).

```
// Fig. 20.15: fig20_15.cpp
// Testing Standard Library vector class template
// element-manipulation functions
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > v( a, a + SIZE );
   ostream_iterator< int > output( cout, " " );
   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output );

   cout << "\nFirst element of v: " << v.front()
        << "\nLast element of v: " << v.back();

   v[ 0 ] = 7;         // set first element to 7
   v.at( 2 ) = 10;     // set element at position 2 to 10
   v.insert( v.begin() + 1, 22 );  // insert 22 as 2nd element
```

Fig. 20.15 Demonstrating Standard Library **vector** class template element-manipulation functions (part 1 of 2).

```
   cout << "\nContents of vector v after changes: ";
   copy( v.begin(), v.end(), output );
```

```
    try {
       v.at( 100 ) = 777;   // access element out of range
    }
    catch ( out_of_range e ) {
       cout << "\nException: " << e.what();
    }

    v.erase( v.begin() );
    cout << "\nContents of vector v after erase: ";
    copy( v.begin(), v.end(), output );
    v.erase( v.begin(), v.end() );
    cout << "\nAfter erase, vector v "
         << ( v.empty() ? "is" : "is not" ) << " empty";

    v.insert( v.begin(), a, a + SIZE );
    cout << "\nContents of vector v before clear: ";
    copy( v.begin(), v.end(), output );
    v.clear();  // clear calls erase to empty a collection
    cout << "\nAfter clear, vector v "
         << ( v.empty() ? "is" : "is not" ) << " empty";

    cout << endl;
    return 0;
 }
```

```
Vector v contains: 1 2 3 4 5 6
First element of v: 1
Last element of v: 6
Contents of vector v after changes: 7 22 2 10 4 5 6
Exception: invalid vector<T> subscript
Contents of vector v after erase: 22 2 10 4 5 6
After erase, vector v is empty
Contents of vector v before clear: 1 2 3 4 5 6
After clear, vector v is empty
```

Fig. 20.15 Demonstrating Standard Library **`vector`** class template element-manipulation functions (part 2 of 2).

STL exception types	Description
`out_of_range`	Indicates when subscript is out of range—e.g., when an invalid subscript is specified to **`vector`** member function **`at`**.
`invalid_argument`	Indicates an invalid argument was passed to a function.
`length_error`	Indicates an attempt to create too long a container, **`string`**, etc.
`bad_alloc`	Indicates that an attempt to allocate memory with **`new`** (or with an allocator) failed because not enough memory was available.

Fig. 20.16 STL exception types.

```
// Fig. 20.17: fig20_17.cpp
// Testing Standard Library class list
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template < class T >
void printList( const list< T > &listRef );

int main()
{
   const int SIZE = 4;
   int a[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values, otherValues;

   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );

   cout << "values contains: ";
   printList( values );
   values.sort();
   cout << "\nvalues after sorting contains: ";
   printList( values );

   otherValues.insert( otherValues.begin(), a, a + SIZE );
```

Fig. 20.17 Demonstrating Standard Library **`list`** class template (part 1 of 3).

```
   cout << "\notherValues contains: ";
   printList( otherValues );
   values.splice( values.end(), otherValues );
   cout << "\nAfter splice values contains: ";
   printList( values );

   values.sort();
   cout << "\nvalues contains: ";
   printList( values );
   otherValues.insert( otherValues.begin(), a, a + SIZE );
   otherValues.sort();
   cout << "\notherValues contains: ";
   printList( otherValues );
   values.merge( otherValues );
   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.pop_front();
   values.pop_back();    // all sequence containers
   cout << "\nAfter pop_front and pop_back values contains:\n";
   printList( values );

   values.unique();
   cout << "\nAfter unique values contains: ";
   printList( values );

```

```
   // method swap is available in all containers
   values.swap( otherValues );
   cout << "\nAfter swap:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.assign( otherValues.begin(), otherValues.end() );
   cout << "\nAfter assign values contains: ";
   printList( values );

   values.merge( otherValues );
   cout << "\nvalues contains: ";
   printList( values );
   values.remove( 4 );
   cout << "\nAfter remove( 4 ) values contains: ";
   printList( values );
   cout << endl;
   return 0;
}
```

Fig. 20.17 Demonstrating Standard Library **list** class template (part 2 of 3).

```
template < class T >
void printList( const list< T > &listRef )
{
   if ( listRef.empty() )
      cout << "List is empty";
   else {
      ostream_iterator< T > output( cout, " " );
      copy( listRef.begin(), listRef.end(), output );
   }
}
```

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
otherValues contains: 2 6 4 8
After splice values contains: 1 2 3 4 2 6 4 8
values contains: 1 2 2 3 4 4 6 8
otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
After pop_front and pop_back values contains:
2 2 2 3 4 4 4 6 6 8
After unique values contains: 2 3 4 6 8
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
After assign values contains: 2 3 4 6 8
values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ) values contains: 2 2 3 3 6 6 8 8
```

Fig. 20.17 Demonstrating Standard Library **list** class template (part 3 of 3).

```
// Fig. 20.18: fig20_18.cpp
// Testing Standard Library class deque
#include <iostream>
#include <deque>
#include <algorithm>

using namespace std;

int main()
{
   deque< double > values;
   ostream_iterator< double > output( cout, " " );

   values.push_front( 2.2 );
   values.push_front( 3.5 );
   values.push_back( 1.1 );

   cout << "values contains: ";

   for ( int i = 0; i < values.size(); ++i )
      cout << values[ i ] << ' ';

   values.pop_front();
   cout << "\nAfter pop_front values contains: ";
   copy ( values.begin(), values.end(), output );

   values[ 1 ] = 5.4;
   cout << "\nAfter values[ 1 ] = 5.4 values contains: ";
   copy ( values.begin(), values.end(), output );
   cout << endl;
   return 0;
}
```

```
values contains: 3.5 2.2 1.1
After pop_front values contains: 2.2 1.1
After values[ 1 ] = 5.4 values contains: 2.2 5.4
```

Fig. 20.18 Demonstrating Standard Library **deque** class template.

```
// Fig. 20.19: fig20_19.cpp
// Testing Standard Library class multiset
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
   typedef multiset< int, less< int > > ims;
   ims intMultiset;    // ims for "integer multiset"
   ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
        << " values of 15 in the multiset\n";
   intMultiset.insert( 15 );
   intMultiset.insert( 15 );
   cout << "After inserts, there are "
```

```
          << intMultiset.count( 15 )
          << " values of 15 in the multiset\n";

     ims::const_iterator result;

     result = intMultiset.find( 15 );  // find returns iterator

     if ( result != intMultiset.end() ) // if iterator not at end
        cout << "Found value 15\n";      // found search value 15

     result = intMultiset.find( 20 );

     if ( result == intMultiset.end() )    // will be true hence
        cout << "Did not find value 20\n"; // did not find 20

     intMultiset.insert( a, a + SIZE ); // add array a to multiset
     cout << "After insert intMultiset contains:\n";
     copy( intMultiset.begin(), intMultiset.end(), output );

     cout << "\nLower bound of 22: "
          << *( intMultiset.lower_bound( 22 ) );
     cout << "\nUpper bound of 22: "
          << *( intMultiset.upper_bound( 22 ) );

     pair< ims::const_iterator, ims::const_iterator > p;

     p = intMultiset.equal_range( 22 );
     cout << "\nUsing equal_range of 22"
          << "\n   Lower bound: " << *( p.first )
          << "\n   Upper bound: " << *( p.second );
```

Fig. 20.19 Demonstrating Standard Library **multiset** class template (part 1 of 2).

```
     cout << endl;
     return 0;
  }
```

```
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset
Found value 15
Did not find value 20
After insert intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100
Lower bound of 22: 22
Upper bound of 22: 30
Using equal_range of 22
   Lower bound: 22
   Upper bound: 30
```

Fig. 20.19 Demonstrating Standard Library **multiset** class template (part 2 of 2).

```
// Fig. 20.20: fig20_20.cpp
// Testing Standard Library class set
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

int main()
{
   typedef set< double, less< double > > double_set;
   const int SIZE = 5;
   double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
   double_set doubleSet( a, a + SIZE );;
   ostream_iterator< double > output( cout, " " );

   cout << "doubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   pair< double_set::const_iterator, bool > p;

   p = doubleSet.insert( 13.8 ); // value not in set
   cout << '\n' << *( p.first )
        << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   p = doubleSet.insert( 9.5 );  // value already in set
   cout << '\n' << *( p.first )
        << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   cout << endl;
   return 0;
}
```

```
doubleSet contains: 2.1 3.7 4.2 9.5
13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
```

Fig. 20.20 Demonstrating Standard Library **set** class template.

```
// Fig. 20.21: fig20_21.cpp
// Testing Standard Library class multimap
#include <iostream>
#include <map>

using namespace std;

int main()
{
   typedef multimap< int, double, less< int > > mmid;
   mmid pairs;

```

```
   cout << "There are currently " << pairs.count( 15 )
        << " pairs with key 15 in the multimap\n";
   pairs.insert( mmid::value_type( 15, 2.7 ) );
   pairs.insert( mmid::value_type( 15, 99.3 ) );
   cout << "After inserts, there are "
        << pairs.count( 15 )
        << " pairs with key 15\n";
   pairs.insert( mmid::value_type( 30, 111.11 ) );
   pairs.insert( mmid::value_type( 10, 22.22 ) );
   pairs.insert( mmid::value_type( 25, 33.333 ) );
   pairs.insert( mmid::value_type( 20, 9.345 ) );
   pairs.insert( mmid::value_type( 5, 77.54 ) );
   cout << "Multimap pairs contains:\nKey\tValue\n";

   for ( mmid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   cout << endl;
   return 0;
}
```

Fig. 20.21 Demonstrating Standard Library **multimap** class template (part 1 of 2).

```
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15
Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

Fig. 20.21 Demonstrating Standard Library **multimap** class template (part 2 of 2).

```
// Fig. 20.22: fig20_22.cpp
// Testing Standard Library class map
#include <iostream>
#include <map>

using namespace std;

int main()
{
   typedef map< int, double, less< int > > mid;
   mid pairs;

   pairs.insert( mid::value_type( 15, 2.7 ) );
   pairs.insert( mid::value_type( 30, 111.11 ) );
   pairs.insert( mid::value_type( 5, 1010.1 ) );
   pairs.insert( mid::value_type( 10, 22.22 ) );
   pairs.insert( mid::value_type( 25, 33.333 ) );
   pairs.insert( mid::value_type( 5, 77.54 ) ); // dupe ignored
```

```
   pairs.insert( mid::value_type( 20, 9.345 ) );
   pairs.insert( mid::value_type( 15, 99.3 ) ); // dupe ignored
   cout << "pairs contains:\nKey\tValue\n";

   mid::const_iterator iter;

   for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   pairs[ 25 ] = 9999.99;  // change existing value for 25
   pairs[ 40 ] = 8765.43;  // insert new value for 40
   cout << "\nAfter subscript operations, pairs contains:"
        << "\nKey\tValue\n";

   for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   cout << endl;
   return 0;
}
```

Fig. 20.22 Demonstrating Standard Library **map** class template (part 1 of 2).

```
pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       33.333
30       111.11

After subscript operations, pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       9999.99
30       111.11
40       8765.43
```

Fig. 20.22 Demonstrating Standard Library **map** class template (part 2 of 2).

```
// Fig. 20.23: fig20_23.cpp
// Testing Standard Library class stack
#include <iostream>
#include <stack>
#include <vector>
#include <list>

using namespace std;

template< class T >
void popElements( T &s );

```

```
int main()
{
   stack< int > intDequeStack; // default is deque-based stack
   stack< int, vector< int > > intVectorStack;
   stack< int, list< int > > intListStack;

   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );
   }

   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );

   cout << endl;
   return 0;
}

```

Fig. 20.23 Demonstrating Standard Library `stack` adapter class (part 1 of 2).

```
template< class T >
void popElements( T &s )
{
   while ( !s.empty() ) {
      cout << s.top() << ' ';
      s.pop();
   }
}
```

```
Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

Fig. 20.23 Demonstrating Standard Library `stack` adapter class (part 2 of 2).

```
// Fig. 20.24: fig20_24.cpp
// Testing Standard Library adapter class template queue
#include <iostream>
#include <queue>

using namespace std;

int main()
{
   queue< double > values;

   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );

   cout << "Popping from values: ";

   while ( !values.empty() ) {
      cout << values.front() << ' ';    // does not remove
      values.pop();                     // removes element
   }

   cout << endl;
   return 0;
}
```

```
Popping from values: 3.2 9.8 5.4
```

Fig. 20.24 Demonstrating Standard Library **queue** adapter class templates.

```
// Fig. 20.25: fig20_25.cpp
// Testing Standard Library class priority_queue
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

int main()
{
   priority_queue< double > priorities;

   priorities.push( 3.2 );
   priorities.push( 9.8 );
   priorities.push( 5.4 );

   cout << "Popping from priorities: ";

   while ( !priorities.empty() ) {
      cout << priorities.top() << ' ';
      priorities.pop();
   }

   cout << endl;
   return 0;
}
```

```
Popping from priorities: 9.8 5.4 3.2
```

Fig. 20.25 Demonstrating Standard Library **`priority_queue`** adapter class.

```
// Fig. 20.26: fig20_26.cpp
// Demonstrating fill, fill_n, generate, and generate_n
// Standard Library methods.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

char nextLetter();

int main()
{
   vector< char > chars( 10 );
   ostream_iterator< char > output( cout, " " );

   fill( chars.begin(), chars.end(), '5' );
   cout << "Vector chars after filling with 5s:\n";
   copy( chars.begin(), chars.end(), output );

   fill_n( chars.begin(), 5, 'A' );
   cout << "\nVector chars after filling five elements"
        << " with As:\n";
   copy( chars.begin(), chars.end(), output );

   generate( chars.begin(), chars.end(), nextLetter );
   cout << "\nVector chars after generating letters A-J:\n";
   copy( chars.begin(), chars.end(), output );
```

Fig. 20.26 Demonstrating Standard Library functions **`fill`**, **`fill_n`**, **`generate`** and **`generate_n`** (part 1 of 2).

```

   generate_n( chars.begin(), 5, nextLetter );
   cout << "\nVector chars after generating K-O for the"
        << " first five elements:\n";
   copy( chars.begin(), chars.end(), output );

   cout << endl;
   return 0;
}

char nextLetter()
{
   static char letter = 'A';
   return letter++;
}
```

```
Vector chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:
A B C D E F G H I J

Vector chars after generating K-O for the first five el-
ements:
K L M N O F G H I J
```

Fig. 20.26 Demonstrating Standard Library functions **fill**, **fill_n**, **generate** and **generate_n** (part 2 of 2).

```
// Fig. 20.27: fig20_27.cpp
// Demonstrates Standard Library functions equal,
// mismatch, lexicographical_compare.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
   vector< int > v1( a1, a1 + SIZE ),
                 v2( a1, a1 + SIZE ),
                 v3( a2, a2 + SIZE );
   ostream_iterator< int > output( cout, " " );

```

Fig. 20.27 Demonstrating Standard Library functions **equal**, **mismatch** and **lexicographical_compare** (part 1 of 2).

```
   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );
   cout << "\nVector v3 contains: ";
   copy( v3.begin(), v3.end(), output );

   bool result = equal( v1.begin(), v1.end(), v2.begin() );
   cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
        << " equal to vector v2.\n";

   result = equal( v1.begin(), v1.end(), v3.begin() );
   cout << "Vector v1 " << ( result ? "is" : "is not" )
        << " equal to vector v3.\n";

   pair< vector< int >::iterator,
         vector< int >::iterator > location;
   location = mismatch( v1.begin(), v1.end(), v3.begin() );
   cout << "\nThere is a mismatch between v1 and v3 at "
```

```
        << "location " << ( location.first - v1.begin() )
        << "\nwhere v1 contains " << *location.first
        << " and v3 contains " << *location.second
        << "\n\n";

   char c1[ SIZE ] = "HELLO", c2[ SIZE ] = "BYE BYE";

   result =
      lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
   cout << c1
        << ( result ? " is less than " : " is greater than " )
        << c2;

   cout << endl;
   return 0;
}
```

```
Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.

There is a mismatch between v1 and v3 at location 4
where v1 contains 5 and v2 contains 1000

HELLO is greater than BYE BYE
```

Fig. 20.27 Demonstrating Standard Library functions **equal**, **mismatch** and **lexicographical_compare** (part 2 of 2).

```
// Fig. 20.28: fig20_28.cpp
// Demonstrates Standard Library functions remove, remove_if
// remove_copy and remove_copy_if
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool greater9( int );

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );

   // Remove 10 from v
   vector< int > v( a, a + SIZE );
   vector< int >::iterator newLastElement;
   cout << "Vector v before removing all 10s:\n";
   copy( v.begin(), v.end(), output );
   newLastElement = remove( v.begin(), v.end(), 10 );
   cout << "\nVector v after removing all 10s:\n";
   copy( v.begin(), newLastElement, output );

   // Copy from v2 to c, removing 10s
```

```
   vector< int > v2( a, a + SIZE );
   vector< int > c( SIZE, 0 );
   cout << "\n\nVector v2 before removing all 10s "
        << "and copying:\n";
   copy( v2.begin(), v2.end(), output );
   remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
   cout << "\nVector c after removing all 10s from v2:\n";
   copy( c.begin(), c.end(), output );

```

Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if** (part 1 of 3).

```
   // Remove elements greater than 9 from v3
   vector< int > v3( a, a + SIZE );
   cout << "\n\nVector v3 before removing all elements"
        << "\ngreater than 9:\n";
   copy( v3.begin(), v3.end(), output );
   newLastElement = remove_if( v3.begin(), v3.end(),
                               greater9 );
   cout << "\nVector v3 after removing all elements"
        << "\ngreater than 9:\n";
   copy( v3.begin(), newLastElement, output );

   // Copy elements from v4 to c,
   // removing elements greater than 9
   vector< int > v4( a, a + SIZE );
   vector< int > c2( SIZE, 0 );
   cout << "\n\nVector v4 before removing all elements"
        << "\ngreater than 9 and copying:\n";
   copy( v4.begin(), v4.end(), output );
   remove_copy_if( v4.begin(), v4.end(),
                   c2.begin(), greater9 );
   cout << "\nVector c2 after removing all elements"
        << "\ngreater than 9 from v4:\n";
   copy( c2.begin(), c2.end(), output );

   cout << endl;
   return 0;
}

bool greater9( int x )
{
   return x > 9;
}
```

Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if** (part 2 of 3).

```
Vector v before removing all 10s:
10 2 10 4 16 6 14 8 12 10
Vector v after removing all 10s:
2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:
10 2 10 4 16 6 14 8 12 10
Vector c after removing all 10s from v2:
2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements
greater than 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 after removing all elements
greater than 9:
2 4 6 8

Vector v4 before removing all elements
greater than 9 and copying:
10 2 10 4 16 6 14 8 12 10
Vector c2 after removing all elements
greater than 9 from v4:
2 4 6 8 0 0 0 0 0 0
```

Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if** (part 3 of 3).

```
// Fig. 20.29: fig20_29.cpp
// Demonstrates Standard Library functions replace, replace_if
// replace_copy and replace_copy_if
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool greater9( int );

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );

   // Replace 10s in v1 with 100
   vector< int > v1( a, a + SIZE );
   cout << "Vector v1 before replacing all 10s:\n";
   copy( v1.begin(), v1.end(), output );
   replace( v1.begin(), v1.end(), 10, 100 );
   cout << "\nVector v1 after replacing all 10s with 100s:\n";
   copy( v1.begin(), v1.end(), output );

```

Fig. 20.29 Demonstrating Standard Library functions **replace**, **replace_if**, **replace_copy** and **replace_copy_if** (part 1 of 3).

```
   // copy from v2 to c1, replacing 10s with 100s
   vector< int > v2( a, a + SIZE );
```

```
   vector< int > c1( SIZE );
   cout << "\n\nVector v2 before replacing all 10s "
        << "and copying:\n";
   copy( v2.begin(), v2.end(), output );
   replace_copy( v2.begin(), v2.end(),
                 c1.begin(), 10, 100 );
   cout << "\nVector c1 after replacing all 10s in v2:\n";
   copy( c1.begin(), c1.end(), output );

   // Replace values greater than 9 in v3 with 100
   vector< int > v3( a, a + SIZE );
   cout << "\n\nVector v3 before replacing values greater"
        << " than 9:\n";
   copy( v3.begin(), v3.end(), output );
   replace_if( v3.begin(), v3.end(), greater9, 100 );
   cout << "\nVector v3 after replacing all values greater"
        << "\nthan 9 with 100s:\n";
   copy( v3.begin(), v3.end(), output );

   // Copy v4 to c2, replacing elements greater than 9 with 100
   vector< int > v4( a, a + SIZE );
   vector< int > c2( SIZE );
   cout << "\n\nVector v4 before replacing all values greater"
        << "\nthan 9 and copying:\n";
   copy( v4.begin(), v4.end(), output );
   replace_copy_if( v4.begin(), v4.end(), c2.begin(),
                    greater9, 100 );
   cout << "\nVector c2 after replacing all values greater"
        << "\nthan 9 in v4:\n";
   copy( c2.begin(), c2.end(), output );

   cout << endl;
   return 0;
}

bool greater9( int x )
{
   return x > 9;
}
```

Fig. 20.29 Demonstrating Standard Library functions **replace**, **replace_if**, **replace_copy** and **replace_copy_if** (part 2 of 3).

```
Vector v1 before replacing all 10s:
10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing all 10s with 100s:
100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:
100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater
than 9 and copying:
10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater
than 9 in v4:
100 2 100 4 100 6 100 8 100 100
```

Fig. 20.29 Demonstrating Standard Library functions **`replace`**, **`replace_if`**, **`replace_copy`** and **`replace_copy_if`** (part 3 of 3).

```
// Fig. 20.30: fig20_30.cpp
// Examples of mathematical algorithms in the Standard Library.
#include <iostream>
#include <algorithm>
#include <numeric>     // accumulate is defined here
#include <vector>

using namespace std;

bool greater9( int );
void outputSquare( int );
int calculateCube( int );

int main()
{
   const int SIZE = 10;
   int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > v( a1, a1 + SIZE );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before random_shuffle: ";
   copy( v.begin(), v.end(), output );
   random_shuffle( v.begin(), v.end() );
   cout << "\nVector v after random_shuffle: ";
   copy( v.begin(), v.end(), output );

   int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   vector< int > v2( a2, a2 + SIZE );
```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 1 of 3).

```
   cout << "\n\nVector v2 contains: ";
```

```
   copy( v2.begin(), v2.end(), output );
   int result = count( v2.begin(), v2.end(), 8 );
   cout << "\nNumber of elements matching 8: " << result;

   result = count_if( v2.begin(), v2.end(), greater9 );
   cout << "\nNumber of elements greater than 9: " << result;

   cout << "\n\nMinimum element in Vector v2 is: "
        << *( min_element( v2.begin(), v2.end() ) );

   cout << "\nMaximum element in Vector v2 is: "
        << *( max_element( v2.begin(), v2.end() ) );

   cout << "\n\nThe total of the elements in Vector v is: "
        << accumulate( v.begin(), v.end(), 0 );

   cout << "\n\nThe square of every integer in Vector v is:\n";
   for_each( v.begin(), v.end(), outputSquare );

   vector< int > cubes( SIZE );
   transform( v.begin(), v.end(), cubes.begin(),
              calculateCube );
   cout << "\n\nThe cube of every integer in Vector v is:\n";
   copy( cubes.begin(), cubes.end(), output );

   cout << endl;
   return 0;
}

bool greater9( int value ) { return value > 9; }

void outputSquare( int value ) { cout << value * value << ' '; }

int calculateCube( int value ) { return value * value * value; }
```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 2 of 3).

```
Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8
```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 3 of 3).

```
// Fig. 20.31: fig20_31.cpp
// Demonstrates search and sort capabilities.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool greater10( int value );

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
   vector< int > v( a, a + SIZE );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output );

   vector< int >::iterator location;
   location = find( v.begin(), v.end(), 16 );

   if ( location != v.end() )
      cout << "\n\nFound 16 at location "
           << ( location - v.begin() );
   else
      cout << "\n\n16 not found";

   location = find( v.begin(), v.end(), 100 );

   if ( location != v.end() )
      cout << "\nFound 100 at location "
           << ( location - v.begin() );
   else
      cout << "\n100 not found";

   location = find_if( v.begin(), v.end(), greater10 );

   if ( location != v.end() )
      cout << "\n\nThe first value greater than 10 is "
           << *location << "\nfound at location "
           << ( location - v.begin() );
   else
      cout << "\n\nNo values greater than 10 were found";

   sort( v.begin(), v.end() );
   cout << "\n\nVector v after sort: ";
   copy( v.begin(), v.end(), output );

```

Fig. 20.31 Basic searching and sorting algorithms of the Standard Library (part 1 of 2).

```
   if ( binary_search( v.begin(), v.end(), 13 ) )
      cout << "\n\n13 was found in v";
   else
      cout << "\n\n13 was not found in v";

   if ( binary_search( v.begin(), v.end(), 100 ) )
      cout << "\n100 was found in v";
   else
      cout << "\n100 was not found in v";

```

```
   cout << endl;
   return 0;
}

bool greater10( int value ) { return value > 10; }
```

```
Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v
```

Fig. 20.31 Basic searching and sorting algorithms of the Standard Library (part 2 of 2).

```
// Fig. 20.32: fig20_32.cpp
// Demonstrates iter_swap, swap and swap_ranges.
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   ostream_iterator< int > output( cout, " " );

   cout << "Array a contains:\n";
   copy( a, a + SIZE, output );
```

Fig. 20.32 Demonstrating **`swap`**, **`iter_swap`** and **`swap_ranges`** (part 1 of 2).

```

   swap( a[ 0 ], a[ 1 ] );
   cout << "\nArray a after swapping a[0] and a[1] "
        << "using swap:\n";
   copy( a, a + SIZE, output );

   iter_swap( &a[ 0 ], &a[ 1 ] );
   cout << "\nArray a after swapping a[0] and a[1] "
        << "using iter_swap:\n";
   copy( a, a + SIZE, output );

   swap_ranges( a, a + 5, a + 5 );
   cout << "\nArray a after swapping the first five elements\n"
        << "with the last five elements:\n";
   copy( a, a + SIZE, output );

   cout << endl;
   return 0;
}
```

```
Array a contains:
1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
6 7 8 9 10 1 2 3 4 5
```

Fig. 20.32 Demonstrating **swap**, **iter_swap** and **swap_ranges** (part 2 of 2).

```
// Fig. 20.33: fig20_33.cpp
// Demonstrates miscellaneous functions: copy_backward, merge,
// unique and reverse.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
   int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE );
   vector< int > v2( a2, a2 + SIZE );

   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );

   vector< int > results( v1.size() );
   copy_backward( v1.begin(), v1.end(), results.end() );
   cout << "\n\nAfter copy_backward, results contains: ";
   copy( results.begin(), results.end(), output );

   vector< int > results2( v1.size() * v2.size() );
   merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
          results2.begin() );
   cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
   copy( results2.begin(), results2.end(), output );

   vector< int >::iterator endLocation;
   endLocation = unique( results2.begin(), results2.end() );
```

Fig. 20.33 Demonstrating **copy_backward**, **merge**, **unique** and **reverse** (part 2 of 2).

```
   cout << "\n\nAfter unique results2 contains:\n";
   copy( results2.begin(), endLocation, output );

   cout << "\n\nVector v1 after reverse: ";
   reverse( v1.begin(), v1.end() );
```

```
   copy( v1.begin(), v1.end(), output );

   cout << endl;
   return 0;
}
```

```
Vector v1 contains: 1 3 5 7 9
Vector v2 contains: 2 4 5 7 9

After copy_backward results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1
```

Fig. 20.33 Demonstrating **copy_backward**, **merge**, **unique** and **reverse** (part 2 of 2).

```
// Fig. 20.34: fig20_34.cpp
// Demonstrates miscellaneous functions: inplace_merge,
// reverse_copy, and unique_copy.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE );

   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );

   inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
   cout << "\nAfter inplace_merge, v1 contains: ";
   copy( v1.begin(), v1.end(), output );

   vector< int > results1;
   unique_copy( v1.begin(), v1.end(),
                back_inserter( results1 ) );
   cout << "\nAfter unique_copy results1 contains: ";
   copy( results1.begin(), results1.end(), output );

   vector< int > results2;
   cout << "\nAfter reverse_copy, results2 contains: ";
   reverse_copy( v1.begin(), v1.end(),
                 back_inserter( results2 ) );
   copy( results2.begin(), results2.end(), output );

   cout << endl;
   return 0;
}
```

```
Vector v1 contains: 1 3 5 7 9 1 3 5 7 9
After inplace_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9
After unique_copy results1 contains: 1 3 5 7 9
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

Fig. 20.34 Demonstrating **inplace_merge**, **unique_copy** and **reverse_copy**.

```
// Fig. 20.35: fig20_35.cpp
// Demonstrates includes, set_difference, set_intersection,
// set_symmetric_difference and set_union.
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
   int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
   int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
   ostream_iterator< int > output( cout, " " );

   cout << "a1 contains: ";
   copy( a1, a1 + SIZE1, output );
   cout << "\na2 contains: ";
   copy( a2, a2 + SIZE2, output );
   cout << "\na3 contains: ";
   copy( a3, a3 + SIZE2, output );

   if ( includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
      cout << "\na1 includes a2";
   else
      cout << "\na1 does not include a2";

   if ( includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
      cout << "\na1 includes a3";
   else
      cout << "\na1 does not include a3";

   int difference[ SIZE1 ];
   int *ptr = set_difference( a1, a1 + SIZE1, a2, a2 + SIZE2,
                              difference );
   cout << "\nset_difference of a1 and a2 is: ";
   copy( difference, ptr, output );

   int intersection[ SIZE1 ];
   ptr = set_intersection( a1, a1 + SIZE1, a2, a2 + SIZE2,
                           intersection );
```

Fig. 20.35 Demonstrating **set** operations of the Standard Library (part 1 of 2).

```
   cout << "\nset_intersection of a1 and a2 is: ";
   copy( intersection, ptr, output );

   int symmetric_difference[ SIZE1 ];
   ptr = set_symmetric_difference( a1, a1 + SIZE1,
```

```
                a2, a2 + SIZE2, symmetric_difference );
   cout << "\nset_symmetric_difference of a1 and a2 is: ";
   copy( symmetric_difference, ptr, output );

   int unionSet[ SIZE3 ];
   ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
   cout << "\nset_union of a1 and a3 is: ";
   copy( unionSet, ptr, output );
   cout << endl;
   return 0;
}
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
a1 includes a2
a1 does not include a3
set_difference of a1 and a2 is: 1 2 3 9 10
set_intersection of a1 and a2 is: 4 5 6 7 8
set_symmetric_difference of a1 and a2 is: 1 2 3 9 10
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

Fig. 20.35 Demonstrating **set** operations of the Standard Library (part 2 of 2).

```
// Fig. 20.36: fig20_36.cpp
// Demonstrates lower_bound, upper_bound and equal_range for
// a sorted sequence of values.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
   vector< int > v( a1, a1 + SIZE );
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains:\n";
   copy( v.begin(), v.end(), output );

   vector< int >::iterator lower;
   lower = lower_bound( v.begin(), v.end(), 6 );
   cout << "\n\nLower bound of 6 is element "
        << ( lower - v.begin() ) << " of vector v";

   vector< int >::iterator upper;
   upper = upper_bound( v.begin(), v.end(), 6 );
   cout << "\nUpper bound of 6 is element "
        << ( upper - v.begin() ) << " of vector v";

   pair< vector< int >::iterator, vector< int >::iterator > eq;
   eq = equal_range( v.begin(), v.end(), 6 );
   cout << "\nUsing equal_range:\n"
        << "   Lower bound of 6 is element "
        << ( eq.first - v.begin() ) << " of vector v";
   cout << "\n   Upper bound of 6 is element "
        << ( eq.second - v.begin() ) << " of vector v";

```

```
    cout << "\n\nUse lower_bound to locate the first point\n"
         << "at which 5 can be inserted in order";
    lower = lower_bound( v.begin(), v.end(), 5 );
    cout << "\n   Lower bound of 5 is element "
         << ( lower - v.begin() ) << " of vector v";

    cout << "\n\nUse upper_bound to locate the last point\n"
         << "at which 7 can be inserted in order";
    upper = upper_bound( v.begin(), v.end(), 7 );
    cout << "\n   Upper bound of 7 is element "
         << ( upper - v.begin() ) << " of vector v";

```

Fig. 20.36 Demonstrating **lower_bound**, **upper_bound** and **equal_range** (part 1 of 2).

```
    cout << "\n\nUse equal_range to locate the first and\n"
         << "last point at which 5 can be inserted in order";
    eq = equal_range( v.begin(), v.end(), 5 );
    cout << "\n   Lower bound of 5 is element "
         << ( eq.first - v.begin() ) << " of vector v";
    cout << "\n   Upper bound of 5 is element "
         << ( eq.second - v.begin() ) << " of vector v"
         << endl;
    return 0;
}
```

```
Vector v contains:
2 2 4 4 4 6 6 6 6 8

Lower bound of 6 is element 5 of vector v
Upper bound of 6 is element 9 of vector v
Using equal_range:
   Lower bound of 6 is element 5 of vector v
   Upper bound of 6 is element 9 of vector v

Use lower_bound to locate the first point
at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v

Use upper_bound to locate the last point
at which 7 can be inserted in order
   Upper bound of 7 is element 9 of vector v

Use equal_range to locate the first and
last point at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v
   Upper bound of 5 is element 5 of vector v
```

Fig. 20.36 Demonstrating **lower_bound**, **upper_bound** and **equal_range** (part 2 of 2).

```
// Fig. 20.37: fig20_37.cpp
// Demonstrating push_heap, pop_heap, make_heap and sort_heap.
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
   int i;
```

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 1 of 3).

```
   vector< int > v( a, a + SIZE ), v2;
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before make_heap:\n";
   copy( v.begin(), v.end(), output );
   make_heap( v.begin(), v.end() );
   cout << "\nVector v after make_heap:\n";
   copy( v.begin(), v.end(), output );
   sort_heap( v.begin(), v.end() );
   cout << "\nVector v after sort_heap:\n";
   copy( v.begin(), v.end(), output );

   // perform the heapsort with push_heap and pop_heap
   cout << "\n\nArray a contains: ";
   copy( a, a + SIZE, output );

   for ( i = 0; i < SIZE; ++i ) {
      v2.push_back( a[ i ] );
      push_heap( v2.begin(), v2.end() );
      cout << "\nv2 after push_heap(a[" << i << "]): ";
      copy( v2.begin(), v2.end(), output );
   }

   for ( i = 0; i < v2.size(); ++i ) {
      cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
      pop_heap( v2.begin(), v2.end() - i );
      copy( v2.begin(), v2.end(), output );
   }

   cout << endl;
   return 0;
}
```

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 2 of 3).

```
Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40
v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 after push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 after push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22
v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100
v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100
```

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 3 of 3).

```
// Fig. 20.38: fig20_38.cpp
// Demonstrating min and max
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   cout << "The minimum of 12 and 7 is: " << min( 12, 7 );
   cout << "\nThe maximum of 12 and 7 is: " << max( 12, 7 );
   cout << "\nThe minimum of 'G' and 'Z' is: "
        << min( 'G', 'Z' );
   cout << "\nThe maximum of 'G' and 'Z' is: "
        << max( 'G', 'Z' ) << endl;
   return 0;
}
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z
```

Fig. 20.38 Demonstrating algorithms **min** and **max**.

Algorithm	Description
adjacent_difference	Beginning with the second element in a sequence, calculate the difference (using operator **-**) between the current element and the previous element, and store the result. The first two input iterator arguments indicate the range of elements in the container and the third output iterator argument indicates where the results should be stored. A second version of this function takes as a fourth argument a binary function to perform a calculation between the current element and the previous element.
inner_product	This function calculates the sum of the products of two sequences by taking corresponding elements in each sequence, multiplying those elements and adding the result to a total.
partial_sum	Calculate a running total (using operator **+**) of the values in a sequence. The first two input iterator arguments indicate the range of elements in the container and the third argument (an output iterator) indicates where the results should be stored. A second version of this function takes as a fourth argument a binary function that performs a calculation between the current value in the sequence and the running total.
nth_element	This function uses three random-access iterators to partition a range of elements. The first and last arguments represent the range of elements. The second argument is the partitioning element's location. After this function executes, all elements to the left of the partitioning element are less than that element and all elements to the right of the partitioning element are greater than or equal to that element. A second version of this function takes as a fourth argument a binary comparison function.
partition	This function is similar to **nth_element**; however, it requires less powerful bidirectional iterators, so it is more flexible than **nth_element**. Function **partition** requires two bidirectional iterators indicating the range of elements to partition. The third element is a unary predicate function that helps partition the elements such that all elements in the sequence for which the predicate is **true** are to the left (toward the beginning of the sequence) of all elements for which the predicate is **false**. A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns **false**.

Fig. 20.39 Algorithms not covered in this chapter.

Algorithm	Description
`stable_partition`	This function is similar to **`partition`** except that elements for which the predicate function returns **`true`** are maintained in their original order and elements for which the **`predicate`** function returns **`false`** are maintained in their original order.
`next_permutation`	Next lexicographical permutation of a sequence.
`prev_permutation`	Previous lexicographical permutation of a sequence.
`rotate`	This function takes three forward iterator arguments and rotates the sequence indicated by the first and last argument by the number of positions indicated by subtracting the first argument from the second argument. For example, the sequence 1, 2, 3, 4, 5 rotated by two positions would be 4, 5, 1, 2, 3.
`rotate_copy`	This function is identical to **`rotate`** except that the results are stored in a separate sequence indicated by the fourth argument—an output iterator. The two sequences must be the same number of elements.
`adjacent_find`	This function returns an input iterator indicating the first of two identical adjacent elements in a sequence. If there are no identical adjacent elements, the iterator is positioned at the **`end`** of the sequence.
`partial_sort`	This function uses three random-access iterators to sort part of a sequence. The first and last arguments indicate the entire sequence of elements. The second argument indicates the ending location for the sorted part of the sequence. By default, elements are ordered using operator **`<`** (a binary predicate function can also be supplied). The elements from the second argument iterator to the end of the sequence are in an undefined order.
`partial_sort_copy`	This function uses two input iterators and two random-access iterators to sort part of the sequence indicated by the two input iterator arguments. The results are stored in the sequence indicated by the two random-access iterator arguments. By default, elements are ordered using operator **`<`** (a binary predicate function can also be supplied). The number of elements sorted is the smaller of the number of elements in the result and the number of elements in the original sequence.
`stable_sort`	The function is similar to **`sort`** except that all equal elements are maintained in their original order.

Fig. 20.39 Algorithms not covered in this chapter.

```
// Fig. 20.40: fig20_40.cpp
// Using a bitset to demonstrate the Sieve of Eratosthenes.
#include <iostream>
#include <iomanip>
#include <bitset>
#include <cmath>

using namespace std;

int main()
{
   const int size = 1024;
   int i, value, counter;
   bitset< size > sieve;

   sieve.flip();

```

Fig. 20.40 Demonstrating class **bitset** and the Sieve of Eratosthenes (part 1 of 3).

```
   // perform Sieve of Eratosthenes
   int finalBit = sqrt( sieve.size() ) + 1;

   for ( i = 2; i < finalBit; ++i )
      if ( sieve.test( i ) )
         for ( int j = 2 * i; j < size; j += i )
            sieve.reset( j );

   cout << "The prime numbers in the range 2 to 1023 are:\n";

   for ( i = 2, counter = 0; i < size; ++i )
      if ( sieve.test( i ) ) {
         cout << setw( 5 ) << i;

         if ( ++counter % 12 == 0 )
            cout << '\n';
      }

   cout << endl;

   // get a value from the user to determine if it is prime
   cout << "\nEnter a value from 1 to 1023 (-1 to end): ";
   cin >> value;

   while ( value != -1 ) {
      if ( sieve[ value ] )
         cout << value << " is a prime number\n";
      else
         cout << value << " is not a prime number\n";

      cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
      cin >> value;
   }

   return 0;
}
```

Fig. 20.40 Demonstrating class **bitset** and the Sieve of Eratosthenes (part 2 of 3).

```
The prime numbers in the range 2 to 1023 are:
    2    3    5    7   11   13   17   19   23   29   31   37
   41   43   47   53   59   61   67   71   73   79   83   89
   97  101  103  107  109  113  127  131  137  139  149  151
  157  163  167  173  179  181  191  193  197  199  211  223
  227  229  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349  353  359
  367  373  379  383  389  397  401  409  419  421  431  433
  439  443  449  457  461  463  467  479  487  491  499  503
  509  521  523  541  547  557  563  569  571  577  587  593
  599  601  607  613  617  619  631  641  643  647  653  659
  661  673  677  683  691  701  709  719  727  733  739  743
  751  757  761  769  773  787  797  809  811  821  823  827
  829  839  853  857  859  863  877  881  883  887  907  911
  919  929  937  941  947  953  967  971  977  983  991  997
 1009 1013 1019 1021

Enter a value from 1 to 1023 (-1 to end): 389
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1
```

Fig. 20.40 Demonstrating class **bitset** and the Sieve of Eratosthenes (part 3 of 3).

STL function objects	Type
divides< T >	arithmetic
equal_to< T >	relational
greater< T >	relational
greater_equal< T >	relational
less< T >	relational
less_equal< T >	relational
logical_and< T >	logical
logical_not< T >	logical
logical_or< T >	logical
minus< T >	arithmetic
modulus< T >	arithmetic
negate< T >	arithmetic
not_equal_to< T >	relational
plus< T >	arithmetic
multiplies< T >	arithmetic

Fig. 20.41 Function objects in the Standard Library .

```
// Fig. 20.42: fig20_42.cpp
// Demonstrating function objects.
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>

using namespace std;

// binary function adds the square of its second argument and
// the running total in its first argument and
// returns the sum
int sumSquares( int total, int value )
   { return total + value * value; }
```

Fig. 20.42 Demonstrating a binary function object (part 1 of 2).

```

// binary function class template which defines an overloaded
// operator() that function adds the square of its second
// argument and the running total in its first argument and
// returns the sum
template< class T >
class SumSquaresClass : public binary_function< T, T, T >
{
public:
   const T &operator()( const T &total, const T &value )
      { return total + value * value; }
};

int main()
{
   const int SIZE = 10;
   int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > v( a1, a1 + SIZE );
   ostream_iterator< int > output( cout, " " );
   int result = 0;

   cout << "vector v contains:\n";
   copy( v.begin(), v.end(), output );
   result = accumulate( v.begin(), v.end(), 0, sumSquares );
   cout << "\n\nSum of squares of elements in vector v using "
        << "binary\nfunction sumSquares: " << result;

   result = accumulate( v.begin(), v.end(), 0,
                        SumSquaresClass< int >() );
   cout << "\n\nSum of squares of elements in vector v using "
        << "binary\nfunction object of type "
        << "SumSquaresClass< int >: " << result << endl;
   return 0;
}
```

```
vector v contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in vector v using binary
function sumSquares: 385

Sum of squares of elements in vector v using binary
function object of type SumSquaresClass< int >: 385
```

Fig. 20.42 Demonstrating a binary function object (part 2 of 2).

Illustrations List (Main Page)

```
// Fig. 21.1: fig21_01.cpp
// Demonstrating data type bool.
#include <iostream>
#include <iomanip>
using namespace std;

```

Fig. 21.1 Demonstrating the fundamental data type **bool** (part 1 of 2).

```
int main()
{
   bool boolean = false;
   int x = 0;

   cout << "boolean is " << boolean
        << "\nEnter an integer: ";
   cin >> x;

   cout << "integer " << x << " is"
        << ( x ? " nonzero " : " zero " )
        << "and interpreted as ";

   if ( x )
      cout << "true\n";
   else
      cout << "false\n";

   boolean = true;
   cout << "boolean is " << boolean;
   cout << "\nboolean output with boolalpha manipulator is "
        << boolalpha << boolean << endl;

   return 0;
}
```

```
boolean is 0
Enter an integer: 22
integer 22 is nonzero and interpreted as true
boolean is 1
boolean output with boolalpha manipulator is true
```

Fig. 21.1 Demonstrating the fundamental data type **bool** (part 2 of 2).

```
// Fig. 21.2: fig21_02.cpp
// Demonstrating the static_cast operator.
#include <iostream.h>

class BaseClass {
public:
   void f( void ) const { cout << "BASE\n"; }
};

class DerivedClass: public BaseClass {
public:
   void f( void ) const { cout << "DERIVED\n"; }
};

void test( BaseClass * );

int main()
{
   // use static_cast for a conversion
   double d = 8.22;
   int x = static_cast< int >( d );

   cout << "d is " << d << "\nx is " << x << endl;

   BaseClass base;  // instantiate base object
   test( &base );   // call test

   return 0;
}

void test( BaseClass *basePtr )
{
   DerivedClass *derivedPtr;

   // cast base class pointer into derived class pointer
   derivedPtr = static_cast< DerivedClass * >( basePtr );
   derivedPtr->f();   // invoke DerivedClass function f
}
```

```
d is 8.22
x is 8
DERIVED
```

Fig. 21.2 Demonstrating operator **static_cast**.

```
// Fig. 21.3: fig21_03.cpp
// Demonstrating the const_cast operator.
#include <iostream.h>

class ConstCastTest {
public:
   void setNumber( int );
   int getNumber() const;
   void printNumber() const;
private:
   int number;
};

void ConstCastTest::setNumber( int num ) { number = num; }
```

```

int ConstCastTest::getNumber() const { return number; }

void ConstCastTest::printNumber() const
{
   cout << "\nNumber after modification: ";

   // the expression number-- would generate compile error
   // undo const-ness to allow modification
   const_cast< ConstCastTest * >( this )->number--;

   cout << number << endl;
}

```

Fig. 21.3 Demonstrating the **`const_cast`** operator (part 1 of 2).

```
int main()
{
   ConstCastTest x;
   x.setNumber( 8 );  // set private data number to 8

   cout << "Initial value of number: " << x.getNumber();

   x.printNumber();
   return 0;
}
```

```
Initial value of number: 8
Number after modification: 7
```

Fig. 21.3 Demonstrating the **`const_cast`** operator (part 2 of 2).

```
// Fig. 21.4: fig21_04.cpp
// Demonstrating reinterpret_cast operator.
#include <iostream.h>

int main()
{
   unsigned x = 22, *unsignedPtr;
   void *voidPtr = &x;
   char *charPtr = "C++";
```

Fig. 21.4 Demonstrating operator **`reinterpret_cast`** (part 1 of 2).

```

   // cast from void * to unsigned *
   unsignedPtr = reinterpret_cast< unsigned * >( voidPtr );

   cout << "*unsignedPtr is " << *unsignedPtr
        << "\ncharPtr is " << charPtr;

   // use reinterpret_cast to cast a char * pointer to unsigned
   cout << "\nchar * to unsigned results in: "
        << ( x = reinterpret_cast< unsigned >( charPtr ) );

   // cast unsigned back to char *
   cout << "\nunsigned to char * results in: "
```

```
         << reinterpret_cast< char * >( x ) << endl;

   return 0;
}
```

```
*unsignedPtr is 22
charPtr is C++
char * to unsigned results in: 4287824
unsigned to char * results in: C++
```

Fig. 21.4 Demonstrating operator **reinterpret_cast** (part 2 of 2).

```
// Fig. 21.5: fig21_05.cpp
// Demonstrating namespaces.
#include <iostream>
using namespace std;  // use std namespace

int myInt = 98;       // global variable

namespace Example {
   const double PI = 3.14159;
   const double E = 2.71828;
   int myInt = 8;
   void printValues();

   namespace Inner {   // nested namespace
      enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
   }
}

namespace {           // unnamed namespace
   double d = 88.22;
}

int main()
{
   // output value d of unnamed namespace
   cout << "d = " << d;

   // output global variable
   cout << "\n(global) myInt = " << myInt;

   // output values of Example namespace
   cout << "\nPI = " << Example::PI << "\nE = "
        << Example::E << "\nmyInt = "
        << Example::myInt << "\nFISCAL3 = "
        << Example::Inner::FISCAL3 << endl;

   Example::printValues();  // invoke printValues function

   return 0;
}

void Example::printValues()
{
   cout << "\n\nIn printValues:\n" << "myInt = "
        << myInt << "\nPI = " << PI << "\nE = "
        << E << "\nd = " << d << "\n(global) myInt = "
        << ::myInt << "\nFISCAL3 = "
        << Inner::FISCAL3 << endl;
```

```
}
```

Fig. 21.5 Demonstrating the use of **namespace**s (part 1 of 2).

```
d = 88.22
(global) myInt = 98
PI = 3.14159
E = 2.71828
myInt = 8
FISCAL3 = 1992


In printValues:
myInt = 8
PI = 3.14159
E = 2.71828
d = 88.22
(global) myInt = 98
FISCAL3 = 1992
```

Fig. 21.5 Demonstrating the use of **namespace**s (part 2 of 2).

```
// Fig. 21.6: fig21_06.cpp
// Demonstrating RTTI capability typeid.
#include <iostream.h>
#include <typeinfo.h>

template < typename T >
T maximum( T value1, T value2, T value3 )
{
   T max = value1;

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   // get the name of the type (i.e., int or double)
   const char *dataType = typeid( T ).name();

   cout << dataType << "s were compared.\nLargest "
        << dataType << " is ";

   return max;
}

int main()
{
   int a = 8, b = 88, c = 22;
   double d = 95.96, e = 78.59, f = 83.89;
```

Fig. 21.6 Demonstrating **typeid** (part 1 of 2).

```

   cout << maximum( a, b, c ) << "\n";
   cout << maximum( d, e, f ) << endl;
```

```

   return 0;
}
```

```
ints were compared.
Largest int is 88
doubles were compared.
Largest double is 95.96
```

Fig. 21.6 Demonstrating **typeid** (part 2 of 2).

```
// Fig. 21.7: fig21_07.cpp
// Demonstrating dynamic_cast.
#include <iostream.h>

const double PI = 3.14159;

```

Fig. 21.7 Demonstrating **dynamic_cast** (part 1 of 3).

```
class Shape {
   public:
      virtual double area() const { return 0.0; }
};

class Circle: public Shape {
public:
   Circle( int r = 1 ) { radius = r; }

   virtual double area() const
   {
      return PI * radius * radius;
   };
protected:
   int radius;
};

class Cylinder: public Circle {
public:
   Cylinder( int h = 1 ) { height = h; }

   virtual double area() const
   {
      return 2 * PI * radius * height +
             2 * Circle::area();
   }
private:
   int height;
};

void outputShapeArea( const Shape * );    // prototype

int main()
{
   Circle circle;
   Cylinder cylinder;
   Shape *ptr = 0;

   outputShapeArea( &circle );    // output circle's area
```

```
   outputShapeArea( &cylinder );  // output cylinder's area
   outputShapeArea( ptr );        // attempt to output area
   return 0;
}

void outputShapeArea( const Shape *shapePtr )
{
   const Circle *circlePtr;
   const Cylinder *cylinderPtr;

   // cast Shape * to a Cylinder *
   cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );
```

Fig. 21.7 Demonstrating **dynamic_cast** (part 2 of 3).

```

   if ( cylinderPtr != 0 )  // if true, invoke area()
      cout << "Cylinder's area: " << cylinderPtr->area();
   else {  // shapePtr does not refer to a cylinder

      // cast shapePtr to a Circle *
      circlePtr = dynamic_cast< const Circle * >( shapePtr );

      if ( circlePtr != 0 )  // if true, invoke area()
         cout << "Circle's area: " << circlePtr->area();
      else
         cout << "Neither a Circle nor a Cylinder.";
   }

   cout << endl;
}
```

```
Circle's area: 3.14159
Cylinder's area: 12.5664
Neither a Circle nor a Cylinder.
```

Fig. 21.7 Demonstrating **dynamic_cast** (part 3 of 3).

.

Operator	Operator keyword	Description
Logical operator keywords		
&&	**and**	logical AND
||	**or**	logical OR
!	**not**	logical NOT
Inequality operator keyword		
!=	**not_eq**	inequality
Bitwise operator keywords		
&	**bitand**	bitwise AND
|	**bitor**	bitwise inclusive OR
^	**xor**	bitwise exclusive OR

Fig. 21.8 Operator keywords as alternatives to operator symbols.

Operator	Operator keyword	Description
~	**compl**	bitwise complement
Bitwise assignment operator keywords		
&=	**and_eq**	bitwise AND assignment
|=	**or_eq**	bitwise inclusive OR assignment
^=	**xor_eq**	bitwise exclusive OR assignment

Fig. 21.8 Operator keywords as alternatives to operator symbols.

```
// Fig. 21.9: fig21_09.cpp
// Demonstrating operator keywords.
#include <iostream>
#include <iomanip>
#include <iso646.h>
using namespace std;

int main()
{
   int a = 8, b = 22;

   cout << boolalpha
        << "   a and b: " << ( a and b )
        << "\n    a or b: " << ( a or b )
        << "\n     not a: " << ( not a )
        << "\na not_eq b: " << ( a not_eq b )
        << "\na bitand b: " << ( a bitand b )
        << "\na bit_or b: " << ( a bitor b )
        << "\n   a xor b: " << ( a xor b )
        << "\n   compl a: " << ( compl a )
        << "\na and_eq b: " << ( a and_eq b )
        << "\n a or_eq b: " << ( a or_eq b )
        << "\na xor_eq b: " << ( a xor_eq b ) << endl;

   return 0;
}
```

```
   a and b: true
    a or b: true
     not a: false
a not_eq b: true
a bitand b: 22
a bit_or b: 22
   a xor b: 0
   compl a: -23
a and_eq b: 22
 a or_eq b: 30
a xor_eq b: 30
```

Fig. 21.9 Demonstrating the use of the operator keywords.

```
// Fig 21.10: array2.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<( ostream &, const Array & );
public:
   Array( int = 10 );  // default/conversion constructor
   ~Array();           // destructor
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
};

#endif
```

Fig. 21.10 Single-argument constructors and implicit conversions (part 1 of 4).

```
// Fig 21.10: array2.cpp
// Member function definitions for class Array
#include <assert.h>
#include "array2.h"

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   cout << "Array constructor called for "
        << size << " elements\n";

   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
}

```

Fig. 21.10 Single-argument constructors and implicit conversions (part 2 of 4).

```
// Destructor for class Array
Array::~Array() { delete [] ptr; }

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ )
      output << a.ptr[ i ] << ' ' ;

   return output;   // enables cout << x << y;
}
```

Fig. 21.10 Single-argument constructors and implicit conversions (part 3 of 4).

```
// Fig 21.10: fig21_10.cpp
// Driver for simple class Array
#include <iostream.h>
#include "array2.h"

void outputArray( const Array & );

int main()
{
   Array integers1( 7 );

   outputArray( integers1 );   // output Array integers1

   outputArray( 15 );  // convert 15 to an Array and output

   return 0;
}

void outputArray( const Array &arrayToOutput )
{
   cout << "The array received contains:\n"
        << arrayToOutput << "\n\n";
}
```

```
Array constructor called for 7 elements
The array received contains:
0 0 0 0 0 0 0

Array constructor called for 15 elements
The array received contains:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Fig. 21.10 Single-argument constructors and implicit conversions (part 4 of 4).

```
// Fig. 21.11: array3.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<( ostream &, const Array & );
public:
   explicit Array( int = 10 );  // default constructor
   ~Array();                    // destructor
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
};

#endif
```

Fig. 21.11 Demonstrating an **`explicit`** constructor (part 1 of 4).

```
// Fig. 21.11: array3.cpp
// Member function definitions for class Array
#include <assert.h>
#include "array3.h"
```

```

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   cout << "Array constructor called for "
        << size << " elements\n";

   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
}

// Destructor for class Array
Array::~Array() { delete [] ptr; }

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ )
      output << a.ptr[ i ] << ' ' ;

   return output;   // enables cout << x << y;
}
```

Fig. 21.11 Demonstrating an **explicit** constructor (part 2 of 4).

```
// Fig. 21.11: fig21_11.cpp
// Driver for simple class Array
#include <iostream.h>
#include "array3.h"

void outputArray( const Array & );

int main()
{
   Array integers1( 7 );

   outputArray( integers1 );   // output Array integers1

   outputArray( 15 );  // convert 15 to an Array and output

```

Fig. 21.11 Demonstrating an **explicit** constructor (part 3 of 4).

```
   outputArray( Array( 15 ) ); // really want to do this!

   return 0;
}

void outputArray( const Array &arrayToOutput )
{
   cout << "The array received contains:\n"
        << arrayToOutput << "\n\n";
}
```

```
Compiling...
Fig21_11.cpp
Fig21_11.cpp(14) : error: 'outputArray' :
   cannot convert parameter 1 from 'const int' to
   'const class Array &'
Array3.cpp
```

Fig. 21.11 Demonstrating an **explicit** constructor (part 4 of 4).

```
// Fig. 21.12: fig21_12.cpp
// Demonstrating storage class specifier mutable.
#include <iostream.h>

class TestMutable {
public:
   TestMutable( int v = 0 ) { value = v; }
   void modifyValue() const { value++; }
   int getValue() const { return value; }
private:
   mutable int value;
};

int main()
{
   const TestMutable t( 99 );

   cout << "Initial value: " << t.getValue();

   t.modifyValue();   // modifies mutable member
   cout << "\nModified value: " << t.getValue() << endl;

   return 0;
}
```

```
Initial value: 99
Modified value: 100
```

Fig. 21.12 Demonstrating a **mutable** data member.

```
// Fig. 21.13: fig21_13.cpp
// Demonstrating operators .* and ->*
#include <iostream.h>

class Test {
public:
   void function() { cout << "function\n"; }
   int value;
};

void arrowStar( Test * );
void dotStar( Test * );

int main()
```

```
{
   Test t;

   t.value = 8;
   arrowStar( &t );
   dotStar( &t );
   return 0;
}

```

Fig. 21.13 Demonstrating the `.*` and `->*` operators (part 1 of 2).

```
void arrowStar( Test *tPtr )
{
   void ( Test::*memPtr )() = &Test::function;
   ( tPtr->*memPtr )();  // invoke function indirectly
}

void dotStar( Test *tPtr )
{
   int Test::*vPtr = &Test::value;
   cout << ( *tPtr ).*vPtr << endl;  // access value
}
```

```
function
8
```

Fig. 21.13 Demonstrating the `.*` and `->*` operators (part 2 of 2).

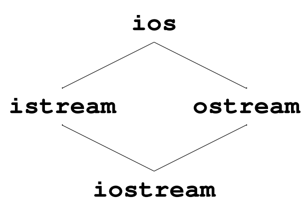


Fig. 21.14 Multiple inheritance to form class **`iostream`**.

```
// Fig. 21.15: fig21_15.cpp
// Attempting to polymorphically call a function
// multiply inherited from two base classes.
#include <iostream.h>

class Base {
public:
   virtual void print() const = 0;  // pure virtual
};

class DerivedOne : public Base {
public:
   // override print function
   void print() const { cout << "DerivedOne\n"; }
};

class DerivedTwo : public Base {
public:
```

```
   // override print function
   void print() const { cout << "DerivedTwo\n"; }
};

class Multiple : public DerivedOne, public DerivedTwo {
public:
   // qualify which version of function print
   void print() const { DerivedTwo::print(); }
};

int main()
{
   Multiple both;    // instantiate Multiple object
   DerivedOne one;   // instantiate DerivedOne object
   DerivedTwo two;   // instantiate DerivedTwo object
```

Fig. 21.15 Attempting to call a multiply inherited function polymorphically (part 1 of 2).

```

   Base *array[ 3 ];
   array[ 0 ] = &both;    // ERROR--ambiguous
   array[ 1 ] = &one;
   array[ 2 ] = &two;

   // polymorphically invoke print
   for ( int k = 0; k < 3; k++ )
      array[ k ] -> print();

   return 0;
}
```

```
Compiling...
fig21_14.cpp
fig21_14.cpp(36) : error: '=' :
   ambiguous conversions from 'class Multiple *' to
   'class Base *'
```

Fig. 21.15 Attempting to call a multiply inherited function polymorphically (part 2 of 2).

```
// Fig. 21.16: fig21_16.cpp
// Using virtual base classes.
#include <iostream.h>

class Base {
public:
   // implicit default constructor

   virtual void print() const = 0; // pure virtual
};

class DerivedOne : virtual public Base {
public:
   // implicit default constructor calls
   // Base default constructor

   // override print function
   void print() const { cout << "DerivedOne\n"; }
};

```

```
class DerivedTwo : virtual public Base {
public:
   // implicit default constructor calls
   // Base default constructor

   // override print function
   void print() const { cout << "DerivedTwo\n"; }
};

class Multiple : public DerivedOne, public DerivedTwo {
public:
   // implicit default constructor calls
   // DerivedOne and DerivedTwo default constructors

   // qualify which version of function print
   void print() const { DerivedTwo::print(); }
};

int main()
{
   Multiple both;   // instantiate Multiple object
   DerivedOne one;  // instantiate DerivedOne object
   DerivedTwo two;  // instantiate DerivedTwo object

   Base *array[ 3 ];
   array[ 0 ] = &both;
   array[ 1 ] = &one;
   array[ 2 ] = &two;

```

Fig. 21.16 Using **`virtual`** base classes (part 1 of 2).

```
   // polymorphically invoke print
   for ( int k = 0; k < 3; k++ )
      array[ k ] -> print();

   return 0;
}
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 21.16 Using **`virtual`** base classes (part 2 of 2).