

C++ HOW TO PROGRAM

SECOND EDITION

Chapter 1	Introduction to Computers and C++ Programming
Chapter 2	Control Structures
Chapter 3	Functions
Chapter 4	Arrays
Chapter 5	Pointers and Strings
Chapter 6	Classes and Data Abstraction
Chapter 7	Classes: Part II
Chapter 8	Operator Overloading
Chapter 9	Inheritance
Chapter 10	Virtual Functions and Polymorphism
Chapter 11	C++ Stream Input/Output
Chapter 12	Templates
Chapter 13	Exception Handling
Chapter 14	File Processing
Chapter 15	Data Structures
Chapter 16	Bits, Characters, Strings, and Structures
Chapter 17	The Preprocessor
Chapter 18	C Legacy Code Topics
Chapter 19	Class string and String Stream Processing
Chapter 20	Standard Template Library (STL)
Chapter 21	ANSI/ISO C++ Standard Language Additions

Illustrations List (Main Page)

- Fig. 1.1** A typical C++ environment.
- Fig. 1.2** Text printing program.
- Fig. 1.3** Some common escape sequences.
- Fig. 1.4** Printing on one line with separate statements using **cout**.
- Fig. 1.5** Printing on multiple lines with a single statement using **cout**.
- Fig. 1.6** An addition program.
- 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.
- Fig. 1.10** Arithmetic operators.
- Fig. 1.11** Precedence of arithmetic operators.
- Fig. 1.12** Order in which a second-degree polynomial is evaluated.
- Fig. 1.13** Equality and relational operators.
- Fig. 1.14** Using equality and relational operators.
- Fig. 1.15** Precedence and associativity of the operators discussed so far.
- Fig. 1.16** Using new-style header files.

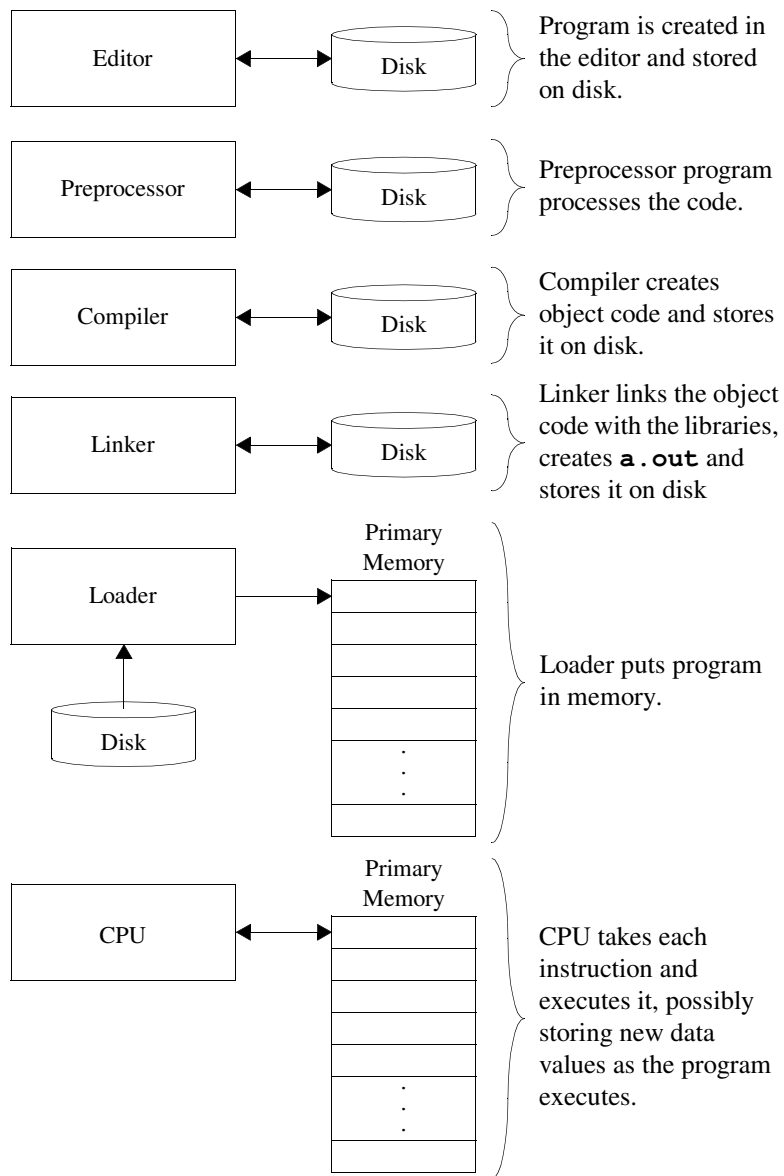


Fig. 1.1 A typical C++ environment.

```
1 // Fig. 1.2: fig01_02.cpp
2 // A first program in C++
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to C++!\n";
8
9     return 0; // indicate that program ended successfully
10 }
```

A screenshot of a terminal window with a black background and white text. The text displayed is "Welcome to C++!".

Fig. 1.2 Text printing program.

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

Fig. 1.3 Some common escape sequences.

```
1 // Fig. 1.4: fig01_04.cpp
2 // Printing a line with multiple statements
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome ";
8     cout << "to C++!\n";
9
10    return 0; // indicate that program ended successfully
11 }
```

A screenshot of a terminal window with a black background and white text. The text displayed is "Welcome to C++!".

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

```
1 // Fig. 1.5: fig01_05.cpp
2 // Printing multiple lines with a single statement
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome\nto\n\nC++!\n";
8
9     return 0; // indicate that program ended successfully
10 }
```




```
Welcome
to

C++!
```

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

```
1 // Fig. 1.6: fig01_06.cpp
2 // Addition program
3 #include <iostream.h>
4
5 int main()
6 {
7     int integer1, integer2, sum; // declaration
8
9     cout << "Enter first integer\n"; // prompt
10    cin >> integer1; // read an integer
11    cout << "Enter second integer\n"; // prompt
12    cin >> integer2; // read an integer
13    sum = integer1 + integer2; // assignment of sum
14    cout << "Sum is " << sum << endl; // print sum
15
16    return 0; // indicate that program ended successfully
17 }
```



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

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

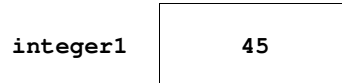


Fig. 1.7 A memory location showing the name and value of a variable.

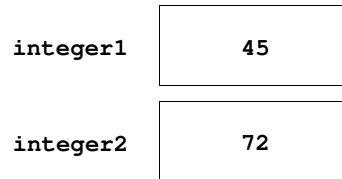


Fig. 1.8 Memory locations after values for two variables have been input.

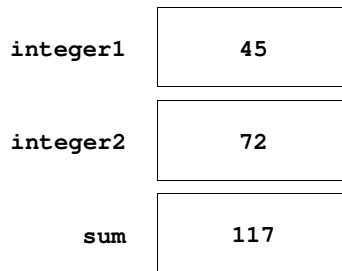


Fig. 1.9 Memory locations after a calculation.

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

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.

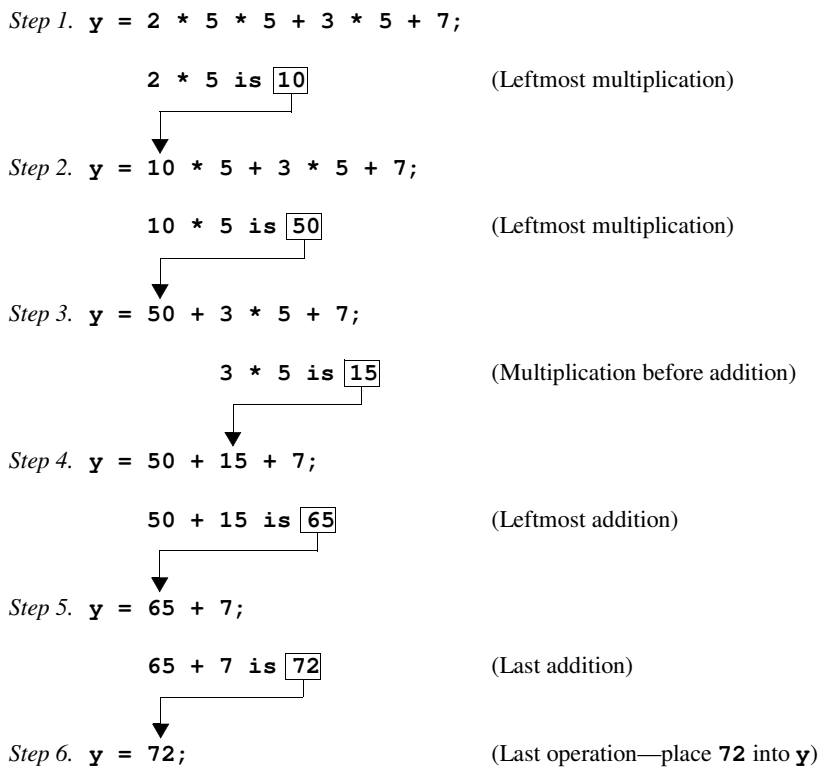


Fig. 1.12 Order in which a second-degree polynomial is evaluated.

Standard algebraic equality operator or relational operator	C++ equality or relational operator	Example of C++ condition	Meaning of C++ condition
<i>Equality operators</i>			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
<i>Relational operators</i>			
>	>	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.

```

1 // Fig. 1.14: fig01_14.cpp
2 // Using if statements, relational
3 // operators, and equality operators
4 #include <iostream.h>
5
6 int main()
7 {
8     int num1, num2;
9
10    cout << "Enter two integers, and I will tell you\n"
11         << "the relationships they satisfy: ";
12    cin >> num1 >> num2;    // read two integers
13
14    if ( num1 == num2 )
15        cout << num1 << " is equal to " << num2 << endl;
16
17    if ( num1 != num2 )
18        cout << num1 << " is not equal to " << num2 << endl;
19
20    if ( num1 < num2 )
21        cout << num1 << " is less than " << num2 << endl;
22
23    if ( num1 > num2 )
24        cout << num1 << " is greater than " << num2 << endl;
25
26    if ( num1 <= num2 )
27        cout << num1 << " is less than or equal to "
28             << num2 << endl;
29
30    if ( num1 >= num2 )
31        cout << num1 << " is greater than or equal to "
32             << num2 << endl;
33
34    return 0;    // indicate that program ended successfully
35 }

```



```

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

```
1 // Fig. 1.16: fig01_16.cpp
2 // Using new-style header files
3 #include <iostream>
4
5 using namespace std;
6
7 int main()
8 {
9     cout << "Welcome to C++!\n";
10    std::cout << "Welcome to C++!\n";
11
12    return 0;    // indicate that program ended successfully
13 }
```



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

Fig. 1.16 Using new-style header files.

Illustrations List (Main Page)

- Fig. 2.1** Flowcharting C++'s sequence structure.
- 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.
- Fig. 2.6** Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.
- Fig. 2.7** C++ program and sample execution for the class average problem with counter-controlled repetition.
- Fig. 2.8** Pseudocode algorithm that uses sentinel-controlled repetition to solve the class average problem.
- Fig. 2.9** C++ program and sample execution for the class average problem with sentinel-controlled repetition.
- Fig. 2.10** Pseudocode for examination results problem.
- Fig. 2.11** C++ program and sample executions for examination results problem.
- Fig. 2.12** Arithmetic assignment operators.
- Fig. 2.13** The increment and decrement operators.
- Fig. 2.14** The difference between preincrementing and postincrementing.
- Fig. 2.15** Precedence of the operators encountered so far in the text.
- Fig. 2.16** Counter-controlled repetition.
- 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** Summation with **for**.
- Fig. 2.21** Calculating compound interest with **for**.
- Fig. 2.22** An example using **switch**.
- Fig. 2.23** The **switch** multiple-selection structure with **breaks**.
- Fig. 2.24** Using the **do/while** structure.
- Fig. 2.25** Flowcharting the **do/while** repetition structure.
- Fig. 2.26** Using the **break** statement in a **for** structure.
- Fig. 2.27** Using the **continue** statement in a **for** structure.
- Fig. 2.28** Truth table for the **&&** (logical AND) operator.
- Fig. 2.29** Truth table for the **||** (logical OR) operator.
- Fig. 2.30** Truth table for operator **!** (logical negation).
- Fig. 2.31** Operator precedence and associativity.
- Fig. 2.32** C++'s single-entry/single-exit sequence, selection, and repetition structures.
- 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.

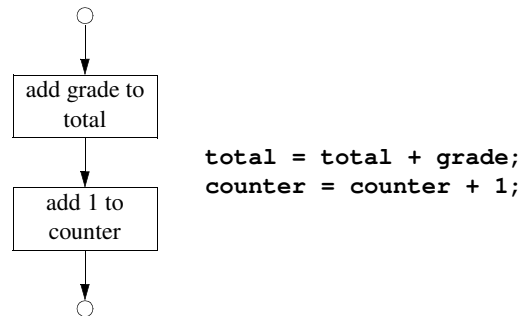


Fig. 2.1 Flowcharting C++'s sequence structure.

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.

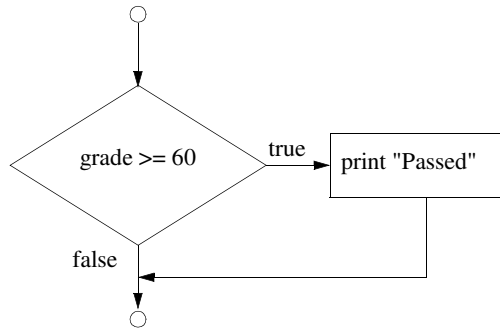


Fig. 2.3 Flowcharting the single-selection **if** structure.

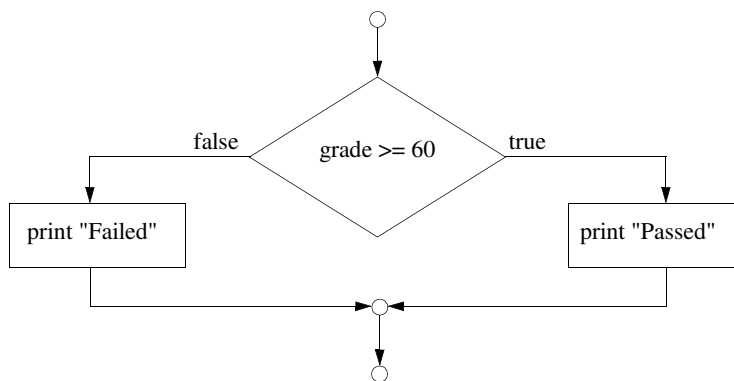


Fig. 2.4 Flowcharting the double-selection **if/else** structure.

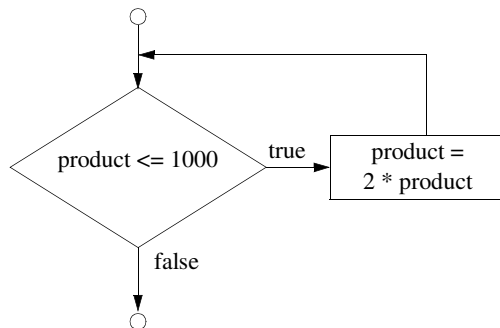


Fig. 2.5 Flowcharting the **while** repetition structure.

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.

```

1 // Fig. 2.7: fig02_07.cpp
2 // Class average program with counter-controlled repetition
3 #include <iostream.h>
4
5 int main()
6 {
7     int total,           // sum of grades
8         gradeCounter,  // number of grades entered
9         grade,         // one grade
10        average;       // average of grades
11
12    // initialization phase
13    total = 0;           // clear total
14    gradeCounter = 1;   // prepare to loop
15
16    // processing phase
17    while ( gradeCounter <= 10 ) { // loop 10 times
18        cout << "Enter grade: "; // prompt for input
19        cin >> grade;           // input grade
20        total = total + grade;  // add grade to total
21        gradeCounter = gradeCounter + 1; // increment counter
22    }
23
24    // termination phase
25    average = total / 10; // integer division
26    cout << "Class average is " << average << endl;
27
28    return 0; // indicate program ended successfully
29 }

```

```

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.

```

1 // Fig. 2.9: fig02_09.cpp
2 // Class average program with sentinel-controlled repetition.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int total,           // sum of grades
9         gradeCounter,  // number of grades entered
10        grade;         // one grade
11    float average;     // number with decimal point for average
12

```

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

```

13 // initialization phase
14 total = 0;
15 gradeCounter = 0;
16
17 // processing phase
18 cout << "Enter grade, -1 to end: ";
19 cin >> grade;
20
21 while ( grade != -1 ) {
22     total = total + grade;
23     gradeCounter = gradeCounter + 1;
24     cout << "Enter grade, -1 to end: ";
25     cin >> grade;
26 }
27
28 // termination phase
29 if ( gradeCounter != 0 ) {
30     average = static_cast< float >( total ) / gradeCounter;
31     cout << "Class average is " << setprecision( 2 )
32         << setiosflags( ios::fixed | ios::showpoint )
33         << average << endl;
34 }
35 else
36     cout << "No grades were entered" << endl;
37
38 return 0; // indicate program ended successfully
39 }

```

```

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.

```

1 // Fig. 2.11: fig02_11.cpp
2 // Analysis of examination results
3 #include <iostream.h>
4
5 int main()
6 {
7     // initialize variables in declarations
8     int passes = 0,           // number of passes
9         failures = 0,        // number of failures
10        studentCounter = 1,   // student counter
11        result;              // one exam result
12
13    // process 10 students; counter-controlled loop
14    while ( studentCounter <= 10 ) {
15        cout << "Enter result (1=pass,2=fail): ";
16        cin >> result;
17

```

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

```

18        if ( result == 1 )           // if/else nested in while
19            passes = passes + 1;
20        else
21            failures = failures + 1;
22
23        studentCounter = studentCounter + 1;
24    }
25
26    // termination phase
27    cout << "Passed " << passes << endl;
28    cout << "Failed " << failures << endl;
29
30    if ( passes > 8 )
31        cout << "Raise tuition " << endl;
32

```

```

33     return 0;    // successful termination
34 }

```

```

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
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
<i>Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
+=	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.

```

1 // Fig. 2.14: fig02_14.cpp
2 // Preincrementing and postincrementing
3 #include <iostream.h>
4
5 int main()
6 {
7     int c;
8
9     c = 5;
10    cout << c << endl;           // print 5
11    cout << c++ << endl;        // print 5 then postincrement
12    cout << c << endl << endl; // print 6
13
14    c = 5;
15    cout << c << endl;           // print 5
16    cout << ++c << endl;        // preincrement then print 6
17    cout << c << endl;           // print 6
18
19    return 0;                   // successful termination
20 }
```

```

5
5
6

5
6
6
```

Fig. 2.14 The difference between preincrementing and postincrementing.

Operators	Associativity	Type
()	left to right	parentheses
++ -- + - <code>static_cast<type>()</code>	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.

```

1 // Fig. 2.16: fig02_16.cpp
2 // Counter-controlled repetition
3 #include <iostream.h>
4
5 int main()
6 {
7     int counter = 1;           // initialization
8
9     while ( counter <= 10 ) { // repetition condition
10         cout << counter << endl;
11         ++counter;           // increment
12     }
13
14     return 0;
15 }
```

Fig. 2.16 Counter-controlled repetition.

```

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

```

1 // Fig. 2.17: fig02_17.cpp
2 // Counter-controlled repetition with the for structure
3 #include <iostream.h>
4
5 int main()
6 {
7     // Initialization, repetition condition, and incrementing
8     // are all included in the for structure header.
9
10    for ( int counter = 1; counter <= 10; counter++ )
11        cout << counter << endl;
12
13    return 0;
14 }

```

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

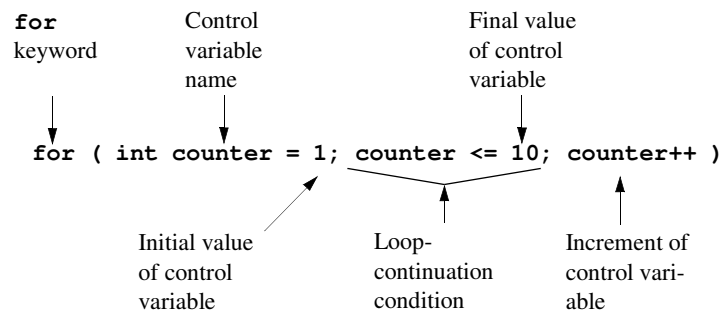


Fig. 2.18 Components of a typical **for** header.

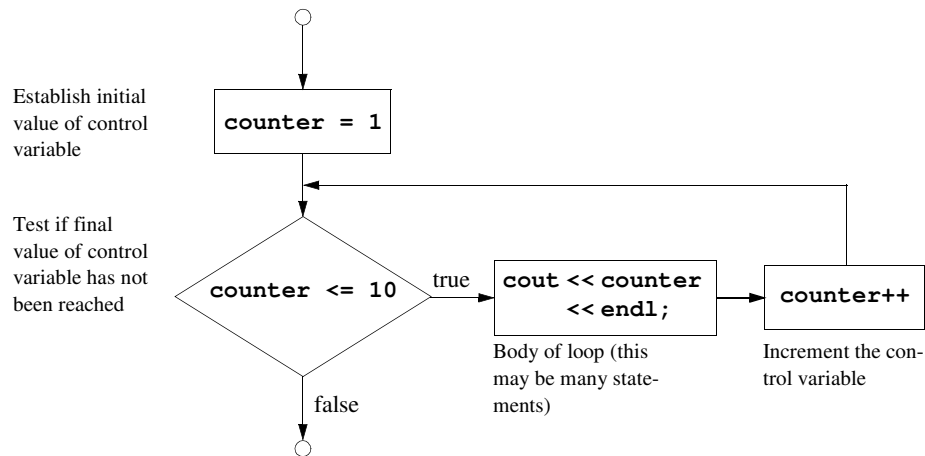


Fig. 2.19 Flowcharting a typical **for** repetition structure.

```

1 // Fig. 2.20: fig02_20.cpp
2 // Summation with for
3 #include <iostream.h>
4
5 int main()
6 {
7     int sum = 0;
8
9     for ( int number = 2; number <= 100; number += 2 )
10         sum += number;
11
12     cout << "Sum is " << sum << endl;
13
14     return 0;
15 }

```

Sum is 2550

Fig. 2.20 Summation with **for**.

```

1 // Fig. 2.21: fig02_21.cpp
2 // Calculating compound interest
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6
7 int main()
8 {
9     double amount,           // amount on deposit
10         principal = 1000.0, // starting principal
11         rate = .05;         // interest rate
12
13     cout << "Year" << setw( 21 )
14         << "Amount on deposit" << endl;
15
16     for ( int year = 1; year <= 10; year++ ) {
17         amount = principal * pow( 1.0 + rate, year );
18         cout << setw( 4 ) << year
19             << setiosflags( ios::fixed | ios::showpoint )
20             << setw( 21 ) << setprecision( 2 )
21             << amount << endl;
22     }
23
24     return 0;
25 }

```

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).

```

1 // Fig. 2.22: fig02_22.cpp
2 // Counting letter grades
3 #include <iostream.h>
4
5 int main()
6 {
7     int grade,          // one grade
8         aCount = 0,    // number of A's
9         bCount = 0,    // number of B's
10        cCount = 0,    // number of C's
11        dCount = 0,    // number of D's
12        fCount = 0;    // number of F's
13
14    cout << "Enter the letter grades." << endl
15         << "Enter the EOF character to end input." << endl;
16
17    while ( ( grade = cin.get() ) != EOF ) {
18
19        switch ( grade ) {          // switch nested in while
20
21            case 'A': // grade was uppercase A
22            case 'a': // or lowercase a
23                ++aCount;
24                break; // necessary to exit switch
25
26            case 'B': // grade was uppercase B
27            case 'b': // or lowercase b
28                ++bCount;
29                break;
30
31            case 'C': // grade was uppercase C
32            case 'c': // or lowercase c
33                ++cCount;
34                break;
35
36            case 'D': // grade was uppercase D
37            case 'd': // or lowercase d
38                ++dCount;
39                break;
40
41            case 'F': // grade was uppercase F
42            case 'f': // or lowercase f
43                ++fCount;
44                break;

```



```

45
46     case '\n': // ignore newlines,
47     case '\t': // tabs,
48     case ' ': // and spaces in input
49         break;
50

```

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

```

51         default: // catch all other characters
52             cout << "Incorrect letter grade entered."
53                 << " Enter a new grade." << endl;
54             break; // optional
55     }
56 }
57
58 cout << "\n\nTotals for each letter grade are:"
59     << "\nA: " << aCount
60     << "\nB: " << bCount
61     << "\nC: " << cCount
62     << "\nD: " << dCount
63     << "\nF: " << fCount << endl;
64
65 return 0;
66 }

```

```

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).

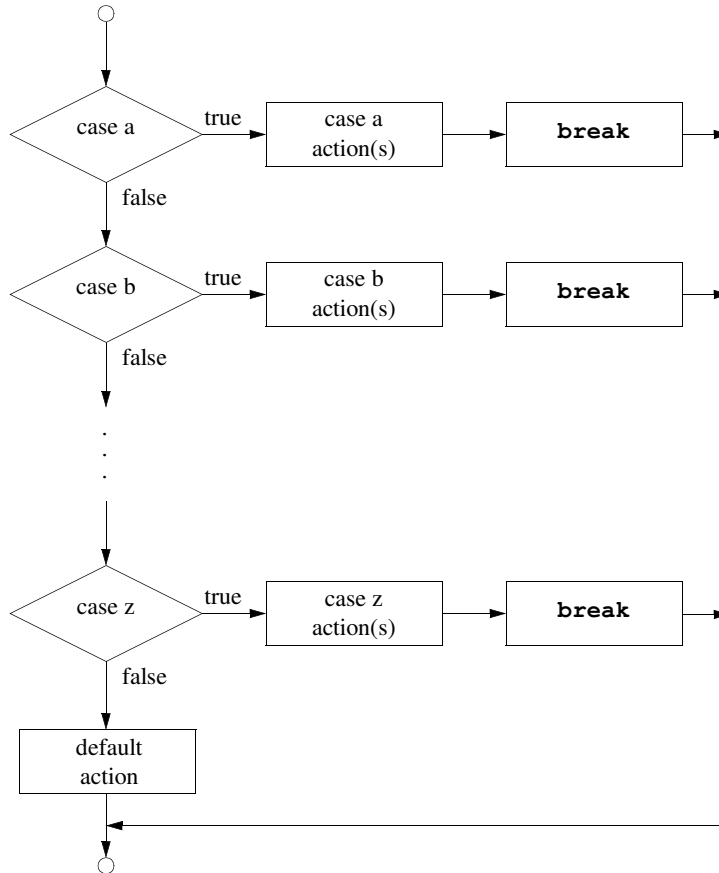


Fig. 2.23 The **switch** multiple-selection structure with **breaks**.

```

1 // Fig. 2.24: fig02_24.cpp
2 // Using the do/while repetition structure
3 #include <iostream.h>
4
5 int main()
6 {
7     int counter = 1;
8
9     do {
10        cout << counter << " ";
11    } while ( ++counter <= 10 );
12
13    cout << endl;
14
15    return 0;
16 }
  
```

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

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

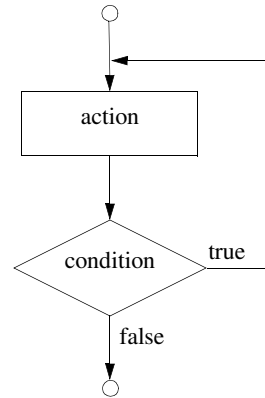


Fig. 2.25 Flowcharting the **do/while** repetition structure.

```

1 // Fig. 2.26: fig02_26.cpp
2 // Using the break statement in a for structure
3 #include <iostream.h>
4
5 int main()
6 {
7     // x declared here so it can be used after the loop
8     int x;
9
10    for ( x = 1; x <= 10; x++ ) {
11
12        if ( x == 5 )
13            break;    // break loop only if x is 5
14
15        cout << x << " ";
16    }
17
18    cout << "\nBroke out of loop at x of " << x << endl;
19    return 0;
20 }
  
```

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).

```

1 // Fig. 2.27: fig02_07.cpp
2 // Using the continue statement in a for structure
3 #include <iostream.h>
4
5 int main()
6 {
7     for ( int x = 1; x <= 10; x++ ) {
8
9         if ( x == 5 )
10            continue; // skip remaining code in loop
11                    // only if x is 5
12
13        cout << x << " ";
14    }
15
16    cout << "\nUsed continue to skip printing the value 5"
17        << endl;
18    return 0;
19 }

```

```

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.

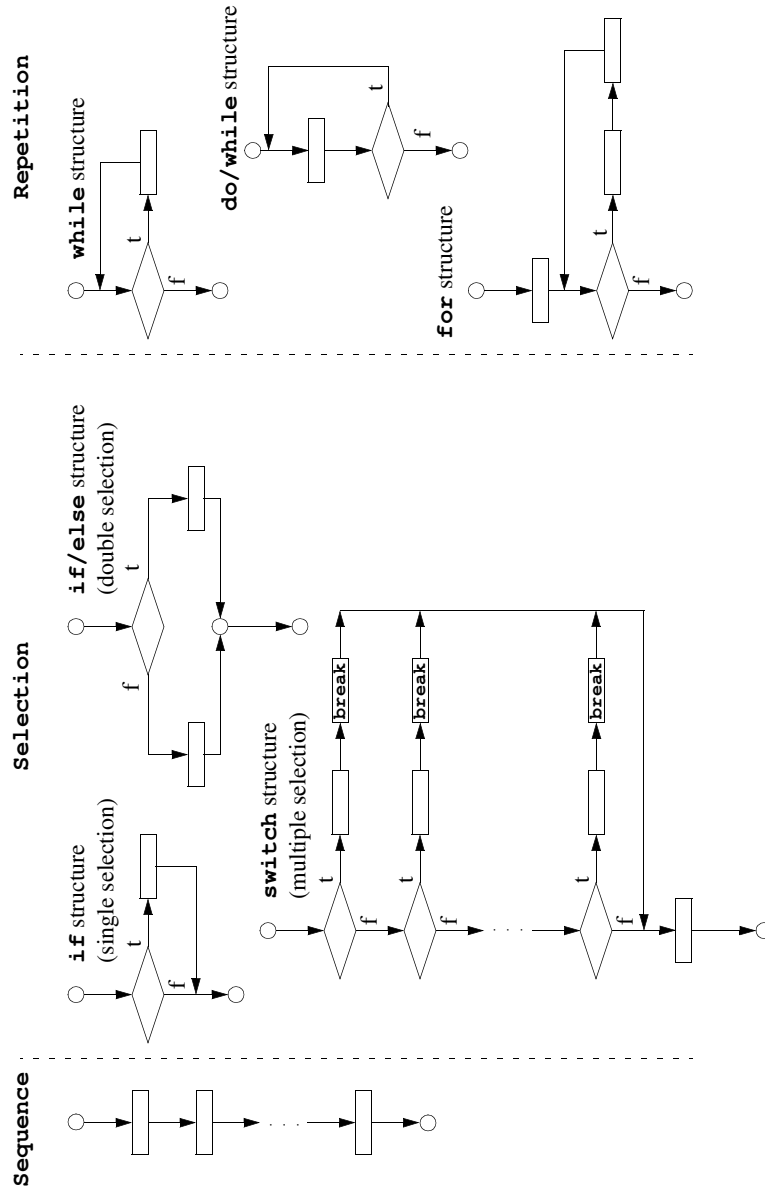


Fig. 2.32 C++'s single-entry/single-exit sequence, selection, and repetition structures.

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.

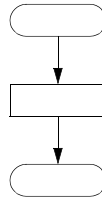


Fig. 2.34 The simplest flowchart.

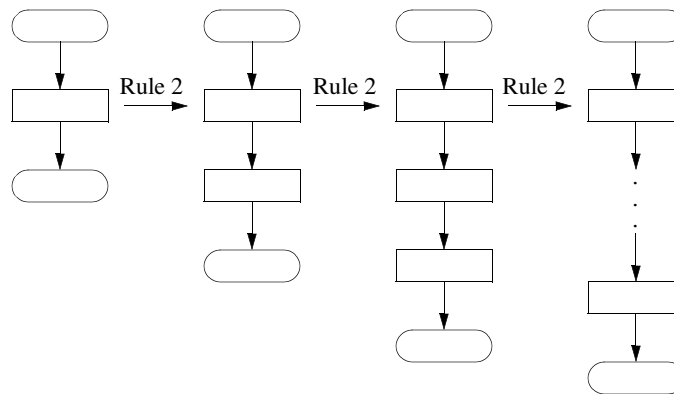


Fig. 2.35 Repeatedly applying rule 2 of Fig. 2.33 to the simplest flowchart.

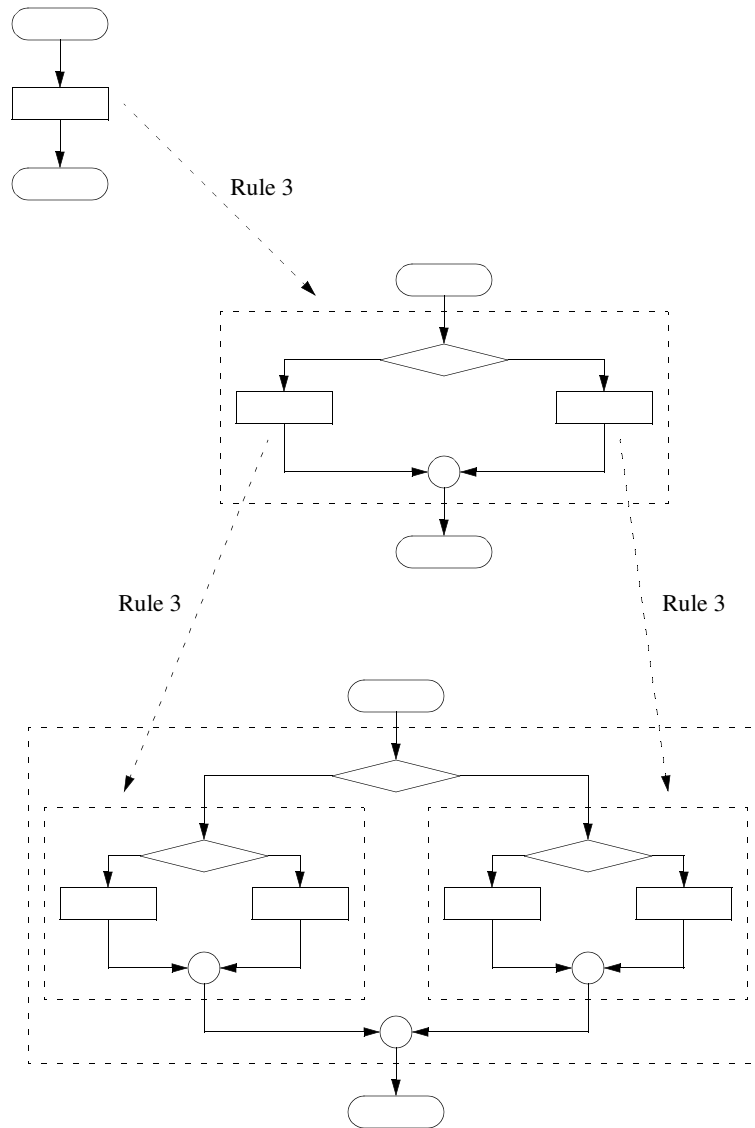


Fig. 2.36 Applying rule 3 of Fig. 2.33 to the simple flowchart.

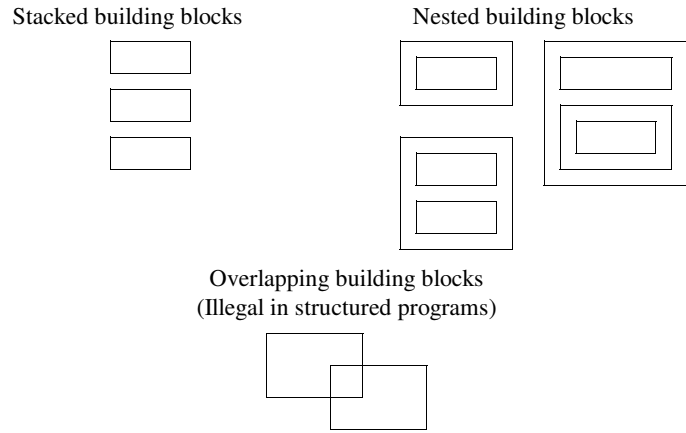


Fig. 2.37 Stacked, nested, and overlapped building blocks.

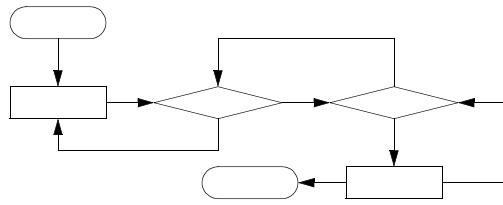


Fig. 2.38 An unstructured flowchart.

Illustrations List (Main Page)

- Fig. 3.1** Hierarchical boss function/worker function relationship.
- Fig. 3.2** Commonly used math library functions
- Fig. 3.3** Creating and using a programmer-defined function
- Fig. 3.4** Programmer-defined **maximum** function (part 1 of 2)
- Fig. 3.5** Promotion hierarchy for built-in data types.
- Fig. 3.6** Standard library header files.
- Fig. 3.7** Shifted, scaled integers produced by `1 + rand() % 6`.
- Fig. 3.8** Rolling a six-sided die 6000 times.
- Fig. 3.9** Randomizing the die-rolling program.
- Fig. 3.10** Program to simulate the game of craps.
- Fig. 3.11** Sample runs for the game of craps.
- Fig. 3.12** A scoping example.
- Fig. 3.13** Recursive evaluation of 5!.
- Fig. 3.14** Calculating factorials with a recursive function.
- Fig. 3.15** Recursively generating Fibonacci numbers.
- Fig. 3.16** Set of recursive calls to method **fibonacci**.
- Fig. 3.17** Summary of recursion examples and exercises in the text.
- Fig. 3.18** Two ways to declare and use functions that take no arguments.
- Fig. 3.19** Using an **inline** function to calculate the volume of a cube.
- Fig. 3.20** An example of call-by-reference.
- Fig. 3.21** Using an initialized reference.
- Fig. 3.22** Attempting to use an uninitialized reference.
- Fig. 3.23** Using default arguments.
- Fig. 3.24** Using the unary scope resolution operator.
- Fig. 3.25** Using overloaded functions.
- Fig. 3.26** Name mangling to enable type-safe linkage.
- Fig. 3.27** Using a function template.

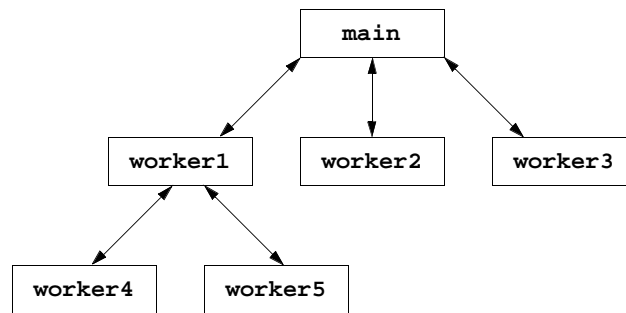


Fig. 3.1 Hierarchical boss function/worker function relationship.

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

Fig. 3.2 Commonly used math library functions.

```

1 // Fig. 3.3: fig03_03.cpp
2 // Creating and using a programmer-defined function
3 #include <iostream.h>
4
5 int square( int ); // function prototype
6
7 int main()
8 {
9     for ( int x = 1; x <= 10; x++ )
10         cout << square( x ) << " ";
11
12     cout << endl;
13     return 0;
14 }
15
16 // Function definition
17 int square( int y )
18 {
19     return y * y;
20 }

```

1 4 9 16 25 36 49 64 81 100

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

```

1 // Fig. 3.4: fig03_04.cpp
2 // Finding the maximum of three integers
3 #include <iostream.h>
4
5 int maximum( int, int, int ); // function prototype
6
7 int main()
8 {
9     int a, b, c;
10
11     cout << "Enter three integers: ";
12     cin >> a >> b >> c;

```

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

```

13
14     // a, b and c below are arguments to
15     // the maximum function call
16     cout << "Maximum is: " << maximum( a, b, c ) << endl;
17
18     return 0;
19 }
20
21 // Function maximum definition
22 // x, y and z below are parameters to
23 // the maximum function definition
24 int maximum( int x, int y, int z )
25 {
26     int max = x;
27
28     if ( y > max )
29         max = y;
30
31     if ( z > max )
32         max = z;
33

```

```

34     return max;
35 }

```

```

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)

<code><assert.h></code>	Contains macros and information for adding diagnostics that aid program debugging. The new version of this header file is <code><cassert></code> .
<code><ctype.h></code>	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 <code><cctype></code> .

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

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

```

1 // Fig. 3.7: fig03_07.cpp
2 // Shifted, scaled integers produced by 1 + rand() % 6
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     for ( int i = 1; i <= 20; i++ ) {
10         cout << setw( 10 ) << ( 1 + rand() % 6 );
11
12         if ( i % 5 == 0 )
13             cout << endl;
14     }
15
16     return 0;
17 }

```

```

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`.

```

1 // Fig. 3.8: fig03_08.cpp
2 // Roll a six-sided die 6000 times
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     int frequency1 = 0, frequency2 = 0,
10        frequency3 = 0, frequency4 = 0,
11        frequency5 = 0, frequency6 = 0,
12        face;
13
14     for ( int roll = 1; roll <= 6000; roll++ ) {
15         face = 1 + rand() % 6;
16
17         switch ( face ) {
18             case 1:
19                 ++frequency1;
20                 break;
21             case 2:
22                 ++frequency2;
23                 break;
24             case 3:
25                 ++frequency3;
26                 break;
27             case 4:
28                 ++frequency4;
29                 break;
30             case 5:
31                 ++frequency5;
32                 break;
33             case 6:
34                 ++frequency6;
35                 break;
36             default:
37                 cout << "should never get here!";
38         }
39     }
40
41     cout << "Face" << setw( 13 ) << "Frequency"
42         << "\n  1" << setw( 13 ) << frequency1
43         << "\n  2" << setw( 13 ) << frequency2
44         << "\n  3" << setw( 13 ) << frequency3
45         << "\n  4" << setw( 13 ) << frequency4
46         << "\n  5" << setw( 13 ) << frequency5
47         << "\n  6" << setw( 13 ) << frequency6 << endl;
48
49     return 0;
50 }

```

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).


```

1 // Fig. 3.9: fig03_09.cpp
2 // Randomizing die-rolling program
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     unsigned seed;
10
11     cout << "Enter seed: ";
12     cin >> seed;
13     srand( seed );
14
15     for ( int i = 1; i <= 10; i++ ) {
16         cout << setw( 10 ) << 1 + rand() % 6;
17
18         if ( i % 5 == 0 )
19             cout << endl;
20     }
21
22     return 0;
23 }

```

The figure shows three separate screenshots of the program's output, each enclosed in a black border. Each screenshot displays the prompt "Enter seed:" followed by a user input and a grid of 10 die rolls arranged in two rows of five. The first screenshot shows a seed of 67, the second shows a seed of 432, and the third shows a seed of 67 again, demonstrating that the same seed produces the same sequence of random numbers.

```

Enter seed: 67
1         6         5         1         2         4
5         6         3         1         2
-----
Enter seed: 432
4         2         6         4         4         3
2         5         1         4         4
-----
Enter seed: 67
1         6         5         1         2         4
5         6         3         1         2

```

Fig. 3.9 Randomizing the die-rolling program.

```

1 // Fig. 3.10: fig03_10.cpp
2 // Craps
3 #include <iostream.h>
4 #include <stdlib.h>
5 #include <time.h>
6
7 int rollDice( void ); // function prototype
8
9 int main()
10 {
11     enum Status { CONTINUE, WON, LOST };
12     int sum, myPoint;
13     Status gameStatus;
14
15     srand( time( NULL ) );
16     sum = rollDice(); // first roll of the dice
17
18     switch ( sum ) {
19         case 7:
20         case 11: // win on first roll
21             gameStatus = WON;
22             break;
23         case 2:

```

```

24     case 3:
25     case 12:           // lose on first roll
26         gameStatus = LOST;
27         break;
28     default:         // remember point
29         gameStatus = CONTINUE;
30         myPoint = sum;
31         cout << "Point is " << myPoint << endl;
32         break;       // optional
33     }
34
35     while ( gameStatus == CONTINUE ) { // keep rolling
36         sum = rollDice();
37
38         if ( sum == myPoint ) // win by making point
39             gameStatus = WON;
40         else
41             if ( sum == 7 ) // lose by rolling 7
42                 gameStatus = LOST;
43     }
44
45     if ( gameStatus == WON )
46         cout << "Player wins" << endl;
47     else
48         cout << "Player loses" << endl;
49
50     return 0;
51 }

```

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

```

52 int rollDice( void )
53 {
54     int die1, die2, workSum;
55
56     die1 = 1 + rand() % 6;
57     die2 = 1 + rand() % 6;
58     workSum = die1 + die2;
59     cout << "Player rolled " << die1 << " + " << die2
60         << " = " << workSum << endl;
61
62     return workSum;
63 }

```

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.

```

1 // Fig. 3.12: fig03_12.cpp
2 // A scoping example
3 #include <iostream.h>
4
5 void a( void ); // function prototype
6 void b( void ); // function prototype
7 void c( void ); // function prototype

```

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

```

8
9 int x = 1; // global variable
10
11 int main()
12 {
13     int x = 5; // local variable to main
14
15     cout << "local x in outer scope of main is " << x << endl;
16
17     { // start new scope
18         int x = 7;
19
20         cout << "local x in inner scope of main is " << x << endl;
21     } // end new scope
22
23     cout << "local x in outer scope of main is " << x << endl;
24
25     a(); // a has automatic local x
26     b(); // b has static local x
27     c(); // c uses global x
28     a(); // a reinitializes automatic local x

```

```
29     b();           // static local x retains its previous value
30     c();           // global x also retains its value
31
32     cout << "local x in main is " << x << endl;
33
34     return 0;
35 }
36
37 void a( void )
38 {
39     int x = 25; // initialized each time a is called
40
41     cout << endl << "local x in a is " << x
42         << " after entering a" << endl;
43     ++x;
44     cout << "local x in a is " << x
45         << " before exiting a" << endl;
46 }
47
48 void b( void )
49 {
50     static int x = 50; // Static initialization only
51                       // first time b is called.
52     cout << endl << "local static x is " << x
53         << " on entering b" << endl;
54     ++x;
55     cout << "local static x is " << x
56         << " on exiting b" << endl;
57 }
58
```

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

```
59 void c( void )
60 {
61     cout << endl << "global x is " << x
62         << " on entering c" << endl;
63     x *= 10;
64     cout << "global x is " << x << " on exiting c" << endl;
65 }
```

```

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).

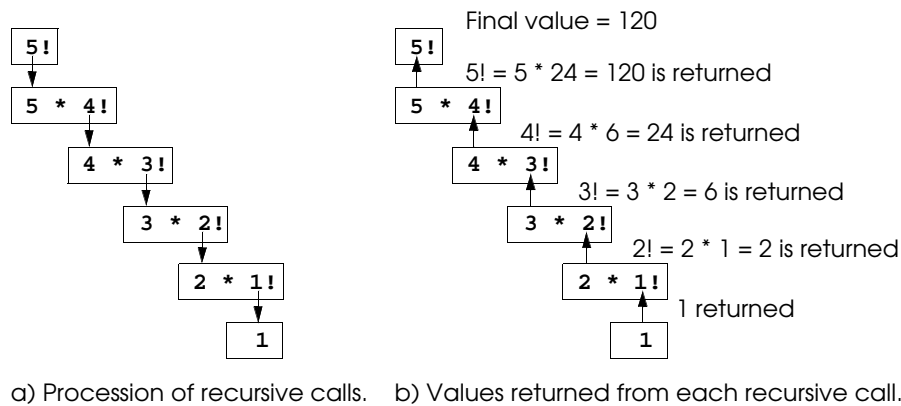


Fig. 3.13 Recursive evaluation of 5!.

```
1 // Fig. 3.14: fig03_14.cpp
2 // Recursive factorial function
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 unsigned long factorial( unsigned long );
7
8 int main()
9 {
10     for ( int i = 0; i <= 10; i++ )
11         cout << setw( 2 ) << i << "! = " << factorial( i ) << endl;
12
13     return 0;
14 }
15
16 // Recursive definition of function factorial
17 unsigned long factorial( unsigned long number )
18 {
19     if ( number <= 1 ) // base case
20         return 1;
21     else // recursive case
22         return number * factorial( number - 1 );
23 }
```

```
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.

```
1 // Fig. 3.15: fig03_15.cpp
2 // Recursive fibonacci function
3 #include <iostream.h>
4
5 long fibonacci( long );
6
7 int main()
8 {
9     long result, number;
10
11     cout << "Enter an integer: ";
12     cin >> number;
13     result = fibonacci( number );
14     cout << "Fibonacci(" << number << ") = " << result << endl;
15     return 0;
16 }
17
18 // Recursive definition of function fibonacci
19 long fibonacci( long n )
20 {
21     if ( n == 0 || n == 1 ) // base case
22         return n;
23     else // recursive case
24         return fibonacci( n - 1 ) + fibonacci( n - 2 );
25 }
```

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).

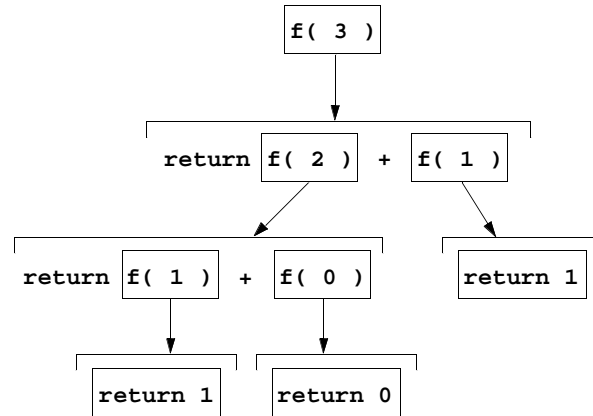


Fig. 3.16 Set of recursive calls to method `fibonacci`.

Chapter	Recursion Examples and Exercises
<i>Chapter 3</i>	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
<i>Chapter 4</i>	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
<i>Chapter 5</i>	Quicksort Maze traversal Printing a string input at the keyboard backwards
<i>Chapter 15</i>	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).

```
1 // Fig. 3.18: fig03_18.cpp
2 // Functions that take no arguments
3 #include <iostream.h>
4
5 void function1();
6 void function2( void );
7
8 int main()
9 {
10     function1();
11     function2();
12
13     return 0;
14 }
15
16 void function1()
17 {
18     cout << "function1 takes no arguments" << endl;
19 }
20
21 void function2( void )
22 {
23     cout << "function2 also takes no arguments" << endl;
24 }
```

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

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

```

1 // Fig. 3.19: fig03_19.cpp
2 // Using an inline function to calculate
3 // the volume of a cube.
4 #include <iostream.h>
5
6 inline float cube( const float s ) { return s * s * s; }
7
8 int main()
9 {
10     cout << "Enter the side length of your cube: ";
11
12     float side;
13
14     cin >> side;
15     cout << "Volume of cube with side "
16         << side << " is " << cube( side ) << endl;
17
18     return 0;
19 }

```

```

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.

```

1 // Fig. 3.20: fig03_20.cpp
2 // Comparing call-by-value and call-by-reference
3 // with references.
4 #include <iostream.h>
5
6 int squareByValue( int );
7 void squareByReference( int & );
8
9 int main()
10 {
11     int x = 2, z = 4;
12
13     cout << "x = " << x << " before squareByValue\n"
14         << "Value returned by squareByValue: "
15         << squareByValue( x ) << endl
16         << "x = " << x << " after squareByValue\n" << endl;
17
18     cout << "z = " << z << " before squareByReference" << endl;
19     squareByReference( z );
20     cout << "z = " << z << " after squareByReference" << endl;
21
22     return 0;
23 }

```

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

```

24
25 int squareByValue( int a )
26 {
27     return a *= a;    // caller's argument not modified
28 }
29
30 void squareByReference( int &cRef )
31 {
32     cRef *= cRef;    // caller's argument modified
33 }

```

```

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).

```

1 // References must be initialized
2 #include <iostream.h>
3
4 int main()
5 {
6     int x = 3, &y = x;    // y is now an alias for x
7
8     cout << "x = " << x << endl << "y = " << y << endl;
9     y = 7;
10    cout << "x = " << x << endl << "y = " << y << endl;
11
12    return 0;
13 }

```

```

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

```

Fig. 3.21 Using an initialized reference.

```

1 // References must be initialized
2 #include <iostream.h>
3
4 int main()
5 {
6     int x = 3, &y;    // Error: y must be initialized
7
8     cout << "x = " << x << endl << "y = " << y << endl;
9     y = 7;
10    cout << "x = " << x << endl << "y = " << y << endl;
11
12    return 0;
13 }

```

```

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

```

Fig. 3.22 Attempting to use an uninitialized reference.

```

1 // Fig. 3.23: fig03_23.cpp
2 // Using default arguments
3 #include <iostream.h>
4
5 int boxVolume( int length = 1, int width = 1, int height = 1 );
6
7 int main()
8 {
9     cout << "The default box volume is: " << boxVolume()
10    << "\n\nThe volume of a box with length 10,\n"
11    << "width 1 and height 1 is: " << boxVolume( 10 )
12    << "\n\nThe volume of a box with length 10,\n"
13    << "width 5 and height 1 is: " << boxVolume( 10, 5 )
14    << "\n\nThe volume of a box with length 10,\n"
15    << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
16    << endl;
17
18    return 0;
19 }
20
21 // Calculate the volume of a box
22 int boxVolume( int length, int width, int height )
23 {
24     return length * width * height;
25 }

```

```

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.

```

1 // Fig. 3.24: fig03_24.cpp
2 // Using the unary scope resolution operator
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 const double PI = 3.14159265358979;
7
8 int main()
9 {
10     const float PI = static_cast< float >( ::PI );
11
12     cout << setprecision( 20 )
13         << " Local float value of PI = " << PI
14         << "\nGlobal double value of PI = " << ::PI << endl;
15
16     return 0;
17 }

```

```

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

```

Fig. 3.24 Using the unary scope resolution operator.

```

1 // Fig. 3.25: fig03_25.cpp
2 // Using overloaded functions
3 #include <iostream.h>
4
5 int square( int x ) { return x * x; }
6
7 double square( double y ) { return y * y; }
8
9 int main()
10 {
11     cout << "The square of integer 7 is " << square( 7 )
12         << "\nThe square of double 7.5 is " << square( 7.5 )
13         << endl;
14
15     return 0;
16 }

```

```

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

```

Fig. 3.25 Using overloaded functions.

```

1 // Name mangling
2 int square(int x) { return x * x; }
3
4 double square(double y) { return y * y; }
5
6 void nothing1(int a, float b, char c, int *d)
7     { } // empty function body
8
9 char *nothing2(char a, int b, float *c, double *d)
10     { return 0; }
11
12 int main()
13 {
14     return 0;
15 }

```

```

public   _main
public   @nothing2$qzqipfpd
public   @nothing1$qifzcpd
public   @square$qd
public   @square$qi

```

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

```

1 // Fig. 3.27: fig03_27.cpp
2 // Using a function template
3 #include <iostream.h>
4
5 template < class T >
6 T maximum( T value1, T value2, T value3 )
7 {
8     T max = value1;
9
10    if ( value2 > max )
11        max = value2;
12
13    if ( value3 > max )
14        max = value3;
15
16    return max;
17 }
18
19 int main()
20 {
21     int int1, int2, int3;
22
23     cout << "Input three integer values: ";
24     cin >> int1 >> int2 >> int3;
25     cout << "The maximum integer value is: "
26         << maximum( int1, int2, int3 );           // int version

```

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

```

27
28     double double1, double2, double3;
29
30     cout << "\nInput three double values: ";
31     cin >> double1 >> double2 >> double3;
32     cout << "The maximum double value is: "
33         << maximum( double1, double2, double3 ); // double version
34
35     char char1, char2, char3;
36
37     cout << "\nInput three characters: ";
38     cin >> char1 >> char2 >> char3;
39     cout << "The maximum character value is: "
40         << maximum( char1, char2, char3 )       // char version
41         << endl;
42
43     return 0;
44 }

```

```
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.
- Fig. 4.2** Operator precedence and associativity.
- Fig. 4.3** Initializing the elements of an array to zeros.
- Fig. 4.4** Initializing the elements of an array with a declaration.
- Fig. 4.5** Generating values to be placed into elements of an array.
- Fig. 4.6** Correctly initializing and using a constant variable.
- Fig. 4.7** A **const** object must be initialized.
- Fig. 4.8** Computing the sum of the elements of an array.
- Fig. 4.9** A student poll analysis program.
- Fig. 4.10** A program that prints histograms.
- Fig. 4.11** Dice-rolling program using arrays instead of **switch**.
- Fig. 4.12** Treating character arrays as strings.
- Fig. 4.13** Comparing **static** array initialization and automatic array initialization.
- Fig. 4.14** Passing arrays and individual array elements to functions.
- Fig. 4.15** Demonstrating the **const** type qualifier.
- Fig. 4.16** Sorting an array with bubble sort.
- Fig. 4.17** Survey data analysis program.
- Fig. 4.18** Sample run for the survey data analysis program.
- Fig. 4.19** Linear search of an array.
- Fig. 4.20** Binary search of a sorted array.
- Fig. 4.21** A double-subscripted array with three rows and four columns.
- Fig. 4.22** Initializing multidimensional arrays.
- Fig. 4.23** Example of using double-subscripted arrays.

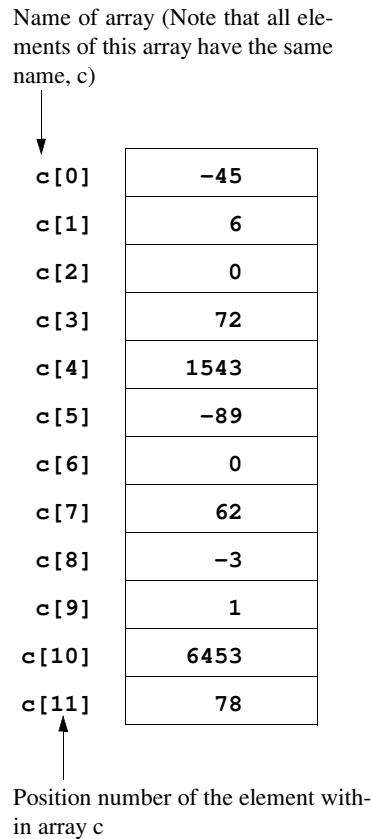


Fig. 4.1 A 12-element array.

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.

```
1 // Fig. 4.3: fig04_03.cpp
2 // initializing an array
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int i, n[ 10 ];
9
10    for ( i = 0; i < 10; i++ )    // initialize array
11        n[ i ] = 0;
12
13    cout << "Element" << setw( 13 ) << "Value" << endl;
14
15    for ( i = 0; i < 10; i++ )    // print array
16        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
17
18    return 0;
19 }
```

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).

```
1 // Fig. 4.4: fig04_04.cpp
2 // Initializing an array with a declaration
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
9
10    cout << "Element" << setw( 13 ) << "Value" << endl;
11
12    for ( int i = 0; i < 10; i++ )
13        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
14
15    return 0;
16 }
```

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.

```

1 // Fig. 4.5: fig04_05.cpp
2 // Initialize array s to the even integers from 2 to 20.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int arraySize = 10;
9     int j, s[ arraySize ];
10
11     for ( j = 0; j < arraySize; j++ )    // set the values
12         s[ j ] = 2 + 2 * j;

```

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

```

13
14     cout << "Element" << setw( 13 ) << "Value" << endl;
15
16     for ( j = 0; j < arraySize; j++ )    // print the values
17         cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
18
19     return 0;
20 }

```

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).

```

1 // Fig. 4.6: fig04_06.cpp
2 // Using a properly initialized constant variable
3 #include <iostream.h>
4
5 int main()
6 {
7     const int x = 7;    // initialized constant variable
8
9     cout << "The value of constant variable x is: "
10         << x << endl;
11
12     return 0;
13 }

```

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).

```
1 // Fig. 4.7: fig04_07.cpp
2 // A const object must be initialized
3
4 int main()
5 {
6     const int x; // Error: x must be initialized
7
8     x = 7;      // Error: cannot modify a const variable
9
10    return 0;
11 }
```

```
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.

```
1 // Fig. 4.8: fig04_08.cpp
2 // Compute the sum of the elements of the array
3 #include <iostream.h>
4
5 int main()
6 {
7     const int arraySize = 12;
8     int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99,
9                          16, 45, 67, 89, 45 };
10    int total = 0;
11
12    for ( int i = 0; i < arraySize ; i++ )
13        total += a[ i ];
14
15    cout << "Total of array element values is " << total << endl;
16    return 0;
17 }
```

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).

```
1 // Fig. 4.9: fig04_09.cpp
2 // Student poll program
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int responseSize = 40, frequencySize = 11;
9     int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
10         10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
11         5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
12     int frequency[ frequencySize ] = { 0 };
13
14     for ( int answer = 0; answer < responseSize; answer++ )
15         ++frequency[ responses[ answer ] ];
16
17     cout << "Rating" << setw( 17 ) << "Frequency" << endl;
18
19     for ( int rating = 1; rating < frequencySize; rating++ )
20         cout << setw( 6 ) << rating
21             << setw( 17 ) << frequency[ rating ] << endl;
22
23     return 0;
24 }
```

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.

```

1 // Fig. 4.10: fig04_10.cpp
2 // Histogram printing program
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     const int arraySize = 10;
9     int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
10
11     cout << "Element" << setw( 13 ) << "Value"
12         << setw( 17 ) << "Histogram" << endl;
13
14     for ( int i = 0; i < arraySize ; i++ ) {
15         cout << setw( 7 ) << i << setw( 13 )
16             << n[ i ] << setw( 9 );
17
18         for ( int j = 0; j < n[ i ]; j++ )    // print one bar
19             cout << '*';
20
21         cout << endl;
22     }
23
24     return 0;
25 }

```

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).

```

1 // Fig. 4.11: fig04_11.cpp
2 // Roll a six-sided die 6000 times
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 int main()
9 {
10     const int arraySize = 7;
11     int face, frequency[ arraySize ] = { 0 };
12
13     srand( time( 0 ) );
14
15     for ( int roll = 1; roll <= 6000; roll++ )
16         ++frequency[ 1 + rand() % 6 ]; // replaces 20-line switch
17                                         // of Fig. 3.8
18
19     cout << "Face" << setw( 13 ) << "Frequency" << endl;
20

```

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

```

21 // ignore element 0 in the frequency array
22 for ( face = 1; face < arraySize ; face++ )
23     cout << setw( 4 ) << face
24         << setw( 13 ) << frequency[ face ] << endl;
25
26 return 0;
27 }

```

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)

```
1 // Fig. 4_12: fig04_12.cpp
2 // Treating character arrays as strings
3 #include <iostream.h>
4
5 int main()
6 {
7     char string1[ 20 ], string2[] = "string literal";
8
9     cout << "Enter a string: ";
10    cin >> string1;
11    cout << "string1 is: " << string1
12         << "\nstring2 is: " << string2
13         << "string1 with spaces between characters is:\n";
14
15    for ( int i = 0; string1[ i ] != '\0'; i++ )
16        cout << string1[ i ] << ' ';
17
18    cin >> string1; // reads "there"
19    cout << "\nstring1 is: " << string1 << endl;
20
21    cout << endl;
22    return 0;
23 }
```

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).

```

1 // Fig. 4.13: fig04_13.cpp
2 // Static arrays are initialized to zero
3 #include <iostream.h>
4
5 void staticArrayInit( void );
6 void automaticArrayInit( void );

```

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

```

7
8 int main()
9 {
10     cout << "First call to each function:\n";
11     staticArrayInit();
12     automaticArrayInit();
13
14     cout << "\n\nSecond call to each function:\n";
15     staticArrayInit();
16     automaticArrayInit();
17     cout << endl;
18
19     return 0;
20 }
21
22 // function to demonstrate a static local array
23 void staticArrayInit( void )
24 {
25     static int array1[ 3 ];
26     int i;
27
28     cout << "\nValues on entering staticArrayInit:\n";
29
30     for ( i = 0; i < 3; i++ )
31         cout << "array1[" << i << "] = " << array1[ i ] << " ";
32
33     cout << "\nValues on exiting staticArrayInit:\n";
34
35     for ( i = 0; i < 3; i++ )
36         cout << "array1[" << i << "] = "
37             << ( array1[ i ] += 5 ) << " ";
38 }
39
40 // function to demonstrate an automatic local array
41 void automaticArrayInit( void )
42 {
43     int i, array2[ 3 ] = { 1, 2, 3 };
44
45     cout << "\n\nValues on entering automaticArrayInit:\n";
46
47     for ( i = 0; i < 3; i++ )
48         cout << "array2[" << i << "] = " << array2[ i ] << " ";
49
50     cout << "\nValues on exiting automaticArrayInit:\n";
51
52     for ( i = 0; i < 3; i++ )
53         cout << "array2[" << i << "] = "
54             << ( array2[ i ] += 5 ) << " ";
55 }

```

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).

```

1 // Fig. 4.14: fig04_14.cpp
2 // Passing arrays and individual array elements to functions
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void modifyArray( int [], int ); // appears strange
7 void modifyElement( int );
8
9 int main()
10 {
11     const int arraySize = 5;
12     int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };
13
14     cout << "Effects of passing entire array call-by-reference:"
15           << "\n\nThe values of the original array are:\n";
16
17     for ( i = 0; i < arraySize; i++ )
18         cout << setw( 3 ) << a[ i ];
19
20     cout << endl;
21
22     // array a passed call-by-reference
23     modifyArray( a, arraySize );
24
25     cout << "The values of the modified array are:\n";
26
27     for ( i = 0; i < arraySize; i++ )
28         cout << setw( 3 ) << a[ i ];
29
30     cout << "\n\n\n"
31           << "Effects of passing array element call-by-value:"

```

```
32         << "\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).

```
33
34     modifyElement( a[ 3 ] );
35
36     cout << "The value of a[3] is " << a[ 3 ] << endl;
37
38     return 0;
39 }
40
41 void modifyArray( int b[], int sizeOfArray )
42 {
43     for ( int j = 0; j < sizeOfArray; j++ )
44         b[ j ] *= 2;
45 }
46
47 void modifyElement( int e )
48 {
49     cout << "Value in modifyElement is "
50         << ( e *= 2 ) << endl;
51 }
```

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).

```
1 // Fig. 4.15: fig04_15.cpp
2 // Demonstrating the const type qualifier
3 #include <iostream.h>
4
5 void tryToModifyArray( const int [] );
6
7 int main()
8 {
9     int a[] = { 10, 20, 30 };
10
11     tryToModifyArray( a );
12     cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
13     return 0;
14 }
15
16 void tryToModifyArray( const int b[] )
17 {
18     b[ 0 ] /= 2;    // error
19     b[ 1 ] /= 2;    // error
20     b[ 2 ] /= 2;    // error
21 }
```

```

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.

```

1 // Fig. 4.16: fig04_16.cpp
2 // This program sorts an array's values into
3 // ascending order
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 int main()
8 {
9     const int arraySize = 10;
10    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11    int i, hold;
12
13    cout << "Data items in original order\n";
14
15    for ( i = 0; i < arraySize; i++ )
16        cout << setw( 4 ) << a[ i ];
17
18    for ( int pass = 0; pass < arraySize - 1; pass++ ) // passes
19
20        for ( i = 0; i < arraySize - 1; i++ ) // one pass
21
22            if ( a[ i ] > a[ i + 1 ] ) { // one comparison
23                hold = a[ i ]; // one swap
24                a[ i ] = a[ i + 1 ];
25                a[ i + 1 ] = hold;
26            }
27
28    cout << "\nData items in ascending order\n";
29

```

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

```

30    for ( i = 0; i < arraySize; i++ )
31        cout << setw( 4 ) << a[ i ];
32
33    cout << endl;
34    return 0;
35 }

```

```

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).

```

1 // Fig. 4.17: fig04_17.cpp
2 // This program introduces the topic of survey data analysis.
3 // It computes the mean, median, and mode of the data.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void mean( const int [], int );
8 void median( int [], int );
9 void mode( int [], int [], int );
10 void bubbleSort( int[], int );
11 void printArray( const int[], int );
12
13 int main()
14 {
15     const int responseSize = 99;
16     int frequency[ 10 ] = { 0 },
17     response[ responseSize ] =
18         { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
19           7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
20           6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
21           7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
22           6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
23           7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
24           5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
25           7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
26           7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
27           4, 5, 6, 1, 6, 5, 7, 8, 7 };
28
29     mean( response, responseSize );
30     median( response, responseSize );
31     mode( frequency, response, responseSize );
32
33     return 0;
34 }
35
36 void mean( const int answer[], int arraySize )
37 {
38     int total = 0;
39
40     cout << "*****\n Mean\n*****\n";
41
42     for ( int j = 0; j < arraySize; j++ )
43         total += answer[ j ];
44

```

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

```

45     cout << "The mean is the average value of the data\n"
46         << "items. The mean is equal to the total of\n"
47         << "all the data items divided by the number\n"
48         << "of data items (" << arraySize
49         << "). The mean value for\nthis run is: "
50         << total << " / " << arraySize << " = "
51         << setiosflags( ios::fixed | ios::showpoint )
52         << setprecision( 4 ) << ( float ) total / arraySize
53         << "\n\n";
54 }
55
56 void median( int answer[], int size )
57 {
58     cout << "\n*****\n Median\n*****\n"

```

```

59         << "The unsorted array of responses is";
60
61     printArray( answer, size );
62     bubbleSort( answer, size );
63     cout << "\n\nThe sorted array is";
64     printArray( answer, size );
65     cout << "\n\nThe median is element " << size / 2
66         << " of\nthe sorted " << size
67         << " element array.\nFor this run the median is "
68         << answer[ size / 2 ] << "\n\n";
69 }
70
71 void mode( int freq[], int answer[], int size )
72 {
73     int rating, largest = 0, modeValue = 0;
74
75     cout << "\n*****\n Mode\n*****\n";
76
77     for ( rating = 1; rating <= 9; rating++ )
78         freq[ rating ] = 0;
79
80     for ( int j = 0; j < size; j++ )
81         ++freq[ answer[ j ] ];
82
83     cout << "Response"<< setw( 11 ) << "Frequency"
84         << setw( 19 ) << "Histogram\n\n" << setw( 56 )
85         << "1    1    2    2\n" << setw( 56 )
86         << "5    0    5    0    5\n\n";
87
88     for ( rating = 1; rating <= 9; rating++ ) {
89         cout << setw( 8 ) << rating << setw( 11 )
90             << freq[ rating ] << "        ";
91
92         if ( freq[ rating ] > largest ) {
93             largest = freq[ rating ];
94             modeValue = rating;
95         }

```

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

```

96
97     for ( int h = 1; h <= freq[ rating ]; h++ )
98         cout << '*';
99
100    cout << '\n';
101 }
102
103 cout << "The mode is the most frequent value.\n"
104     << "For this run the mode is " << modeValue
105     << " which occurred " << largest << " times." << endl;
106 }
107
108 void bubbleSort( int a[], int size )
109 {
110     int hold;
111
112     for ( int pass = 1; pass < size; pass++ )
113
114         for ( int j = 0; j < size - 1; j++ )
115
116             if ( a[ j ] > a[ j + 1 ] ) {
117                 hold = a[ j ];
118                 a[ j ] = a[ j + 1 ];

```

```
119         a[ j + 1 ] = hold;
120     }
121 }
122
123 void printArray( const int a[], int size )
124 {
125     for ( int j = 0; j < size; j++ ) {
126         if ( j % 20 == 0 )
127             cout << endl;
128         cout << setw( 2 ) << a[ j ];
129     }
130 }
131 }
132 }
```

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 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 8 8
9 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.

```
1 // Fig. 4.19: fig04_19.cpp
2 // Linear search of an array
3 #include <iostream.h>
4
5 int linearSearch( const int [], int, int );
6
7 int main()
8 {
9     const int arraySize = 100;
10    int a[ arraySize ], searchKey, element;
11
12    for ( int x = 0; x < arraySize; x++ ) // create some data
13        a[ x ] = 2 * x;
14
15    cout << "Enter integer search key:" << endl;
16    cin >> searchKey;
17    element = linearSearch( a, searchKey, arraySize );
18
19    if ( element != -1 )
20        cout << "Found value in element " << element << endl;
21    else
22        cout << "Value not found" << endl;
23
24    return 0;
25 }
```

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

```
26
27 int linearSearch( const int array[], int key, int sizeOfArray )
28 {
29     for ( int n = 0; n < sizeOfArray; n++ )
30         if ( array[ n ] == key )
31             return n;
32
33     return -1;
34 }
```

```
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).

```

1 // Fig. 4.20: fig04_20.cpp
2 // Binary search of an array
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int binarySearch( int [], int, int, int, int );
7 void printHeader( int );
8 void printRow( int [], int, int, int, int );
9
10 int main()
11 {
12     const int arraySize = 15;
13     int a[ arraySize ], key, result;
14
15     for ( int i = 0; i < arraySize; i++ )
16         a[ i ] = 2 * i; // place some data in array
17
18     cout << "Enter a number between 0 and 28: ";
19     cin >> key;
20
21     printHeader( arraySize );
22     result = binarySearch( a, key, 0, arraySize - 1, arraySize );
23

```

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

```

24     if ( result != -1 )
25         cout << '\n' << key << " found in array element "
26             << result << endl;
27     else
28         cout << '\n' << key << " not found" << endl;
29
30     return 0;
31 }
32
33 // Binary search
34 int binarySearch( int b[], int searchKey, int low, int high,
35                 int size )
36 {
37     int middle;
38
39     while ( low <= high ) {
40         middle = ( low + high ) / 2;
41
42         printRow( b, low, middle, high, size );
43
44         if ( searchKey == b[ middle ] ) // match
45             return middle;
46         else if ( searchKey < b[ middle ] )
47             high = middle - 1; // search low end of array
48         else
49             low = middle + 1; // search high end of array
50     }
51
52     return -1; // searchKey not found
53 }
54
55 // Print a header for the output
56 void printHeader( int size )
57 {
58     cout << "\nSubscripts:\n";
59     for ( int i = 0; i < size; i++ )

```

```

60     cout << setw( 3 ) << i << ' ';
61
62     cout << '\n';
63
64     for ( i = 1; i <= 4 * size; i++ )
65         cout << '-';
66
67     cout << endl;
68 }
69

```

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

```

70 // Print one row of output showing the current
71 // part of the array being processed.
72 void printRow( int b[], int low, int mid, int high, int size )
73 {
74     for ( int i = 0; i < size; i++ )
75         if ( i < low || i > high )
76             cout << "      ";
77         else if ( i == mid )           // mark middle value
78             cout << setw( 3 ) << b[ i ] << '*';
79         else
80             cout << setw( 3 ) << b[ i ] << ' ';
81
82     cout << endl;
83 }

```

```

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).

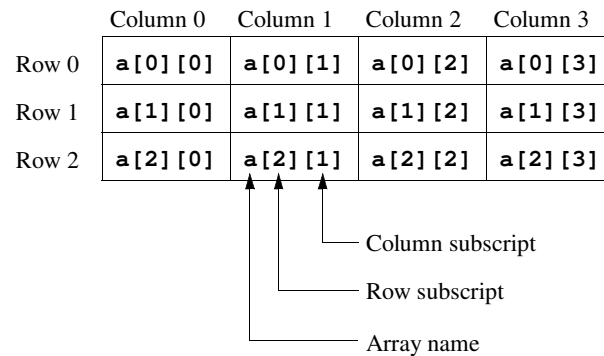


Fig. 4.21 A double-subscripted array with three rows and four columns.

```

1 // Fig. 4.22: fig04_22.cpp
2 // Initializing multidimensional arrays
3 #include <iostream.h>
4
5 void printArray( int [][] [ 3 ] );
6
7 int main()
8 {
9     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
10     array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 },
11     array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
12
13     cout << "Values in array1 by row are:" << endl;
14     printArray( array1 );
15
16     cout << "Values in array2 by row are:" << endl;
17     printArray( array2 );
18
19     cout << "Values in array3 by row are:" << endl;
20     printArray( array3 );
21
22     return 0;
23 }
24

```

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

```

25 void printArray( int a[][ 3 ] )
26 {
27     for ( int i = 0; i < 2; i++ ) {
28
29         for ( int j = 0; j < 3; j++ )
30             cout << a[ i ][ j ] << ' ';
31
32         cout << endl;
33     }
34 }

```

```

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).

```

1 // Fig. 4.23: fig04_23.cpp
2 // Double-subscripted array example
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 const int students = 3; // number of students
7 const int exams = 4; // number of exams
8
9 int minimum( int [][] exams, int, int );
10 int maximum( int [][] exams, int, int );
11 float average( int [], int );
12 void printArray( int [][] exams, int, int );
13

```

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

```

14 int main()
15 {
16     int studentGrades[ students ][ exams ] =
17         { { 77, 68, 86, 73 },
18           { 96, 87, 89, 78 },
19           { 70, 90, 86, 81 } };
20
21     cout << "The array is:\n";
22     printArray( studentGrades, students, exams );
23     cout << "\n\nLowest grade: "
24         << minimum( studentGrades, students, exams )
25         << "\n\nHighest grade: "
26         << maximum( studentGrades, students, exams ) << '\n';
27
28     for ( int person = 0; person < students; person++ )
29         cout << "The average grade for student " << person << " is "
30             << setiosflags( ios::fixed | ios::showpoint )
31             << setprecision( 2 )
32             << average( studentGrades[ person ], exams ) << endl;
33

```

```

34     return 0;
35 }
36
37 // Find the minimum grade
38 int minimum( int grades[][ exams ], int pupils, int tests )
39 {
40     int lowGrade = 100;
41
42     for ( int i = 0; i < pupils; i++ )
43
44         for ( int j = 0; j < tests; j++ )
45
46             if ( grades[ i ][ j ] < lowGrade )
47                 lowGrade = grades[ i ][ j ];
48
49     return lowGrade;
50 }
51
52 // Find the maximum grade
53 int maximum( int grades[][ exams ], int pupils, int tests )
54 {
55     int highGrade = 0;
56
57     for ( int i = 0; i < pupils; i++ )
58
59         for ( int j = 0; j < tests; j++ )
60

```

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

```

61         if ( grades[ i ][ j ] > highGrade )
62             highGrade = grades[ i ][ j ];
63
64     return highGrade;
65 }
66
67 // Determine the average grade for a particular student
68 float average( int setOfGrades[], int tests )
69 {
70     int total = 0;
71
72     for ( int i = 0; i < tests; i++ )
73         total += setOfGrades[ i ];
74
75     return ( float ) total / tests;
76 }
77
78 // Print the array
79 void printArray( int grades[][ exams ], int pupils, int tests )
80 {
81     cout << "                [0] [1] [2] [3]";
82
83     for ( int i = 0; i < pupils; i++ ) {
84         cout << "\nstudentGrades[" << i << "] ";
85
86         for ( int j = 0; j < tests; j++ )
87             cout << setw( 5 ) << grades[ i ][ j ];
88     }
89 }
90 }

```

```
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 The **&** and ***** pointer operators.
Fig. 5.5 Operator precedence and associativity.
Fig. 5.6 Cube a variable using call-by-value.
Fig. 5.7 Cube a variable using call-by-reference with a pointer argument.
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 Converting a string to uppercase.
Fig. 5.11 Printing a string one character at a time using a non-constant pointer to constant data.
Fig. 5.12 Attempting to modify data through a non-constant pointer to constant data.
Fig. 5.13 Attempting to modify a constant pointer to non-constant data.
Fig. 5.14 Attempting to modify a constant pointer to constant data.
Fig. 5.15 Bubble sort with call-by-reference.
Fig. 5.16 The **sizeof** operator when applied to an array name returns the number of bytes in the array.
Fig. 5.17 Using the **sizeof** operator to determine standard data type sizes.
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 Using four methods of referencing array elements.
Fig. 5.21 Copying a string using array notation and pointer notation.
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 Card shuffling and dealing program.
Fig. 5.25 Sample run of card shuffling and dealing program.
Fig. 5.26 Multipurpose sorting program using function pointers.
Fig. 5.27 The outputs of the bubble sort program in Fig. 5.26.
Fig. 5.28 Demonstrating an array of pointers to functions.
Fig. 5.29 The string manipulation functions of the string handling library.
Fig. 5.30 Using **strcpy** and **strncpy**.
Fig. 5.31 Using **strcat** and **strncat**.
Fig. 5.32 Using **strcmp** and **strncmp**.
Fig. 5.33 Using **strtok**.
Fig. 5.34 Using **strlen**.

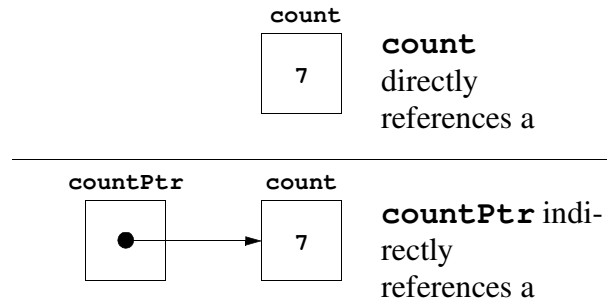


Fig. 5.1 Directly and indirectly referencing a variable.

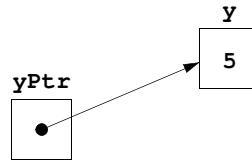


Fig. 5.2 Graphical representation of a pointer pointing to an integer variable in memory.



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

```

1 // Fig. 5.4: fig05_04.cpp
2 // Using the & and * operators
3 #include <iostream.h>
4
5 int main()
6 {
7     int a;        // a is an integer
8     int *aPtr;   // aPtr is a pointer to an integer
9
10    a = 7;
11    aPtr = &a;    // aPtr set to address of a
12
13    cout << "The address of a is " << &a
14         << "\nThe value of aPtr is " << aPtr;
15
16    cout << "\n\nThe value of a is " << a
17         << "\nThe value of *aPtr is " << *aPtr;
18
19    cout << "\n\nShowing that * and & are inverses of "
20         << "each other.\n&*aPtr = " << &*aPtr
21         << "\n*&aPtr = " << *&aPtr << endl;
22    return 0;
23 }

```

```

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 .

```

1 // Fig. 5.6: fig015_06.cpp
2 // Cube a variable using call-by-value
3 #include <iostream.h>
4
5 int cubeByValue( int ); // prototype
6
7 int main()
8 {
9     int number = 5;
10
11     cout << "The original value of number is " << number;
12     number = cubeByValue( number );
13     cout << "\nThe new value of number is " << number << endl;
14     return 0;
15 }
16
17 int cubeByValue( int n )
18 {
19     return n * n * n; // cube local variable n
20 }

```

```

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

```

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

```

1 // Fig. 5.7: fig05_07.cpp
2 // Cube a variable using call-by-reference
3 // with a pointer argument
4 #include <iostream.h>
5
6 void cubeByReference( int * ); // prototype
7
8 int main()
9 {
10     int number = 5;
11
12     cout << "The original value of number is " << number;

```

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

```

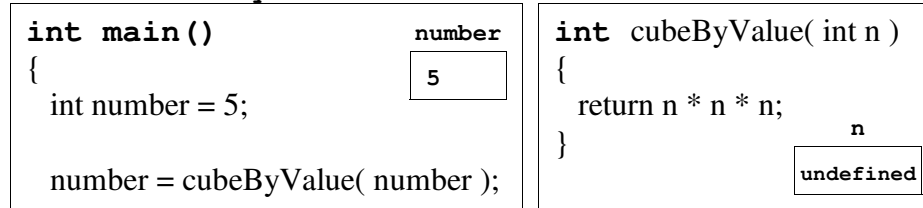
13     cubeByReference( &number );
14     cout << "\nThe new value of number is " << number << endl;
15     return 0;
16 }
17
18 void cubeByReference( int *nPtr )
19 {
20     *nPtr = *nPtr * *nPtr * *nPtr; // cube number in main
21 }

```

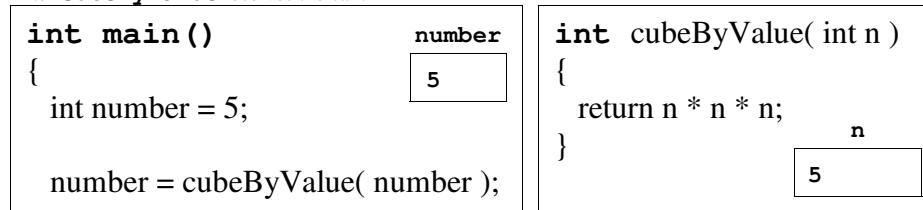
The original value of number is 5
The new value of number is 125

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

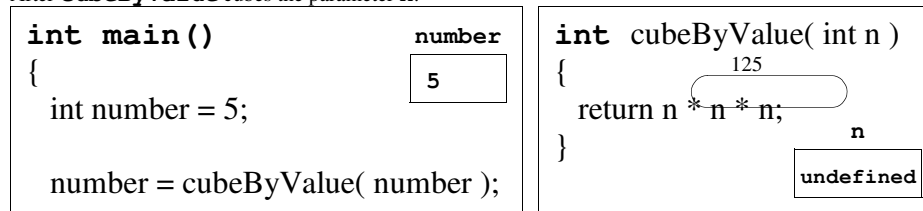
Before **main** calls **cubeByValue**:



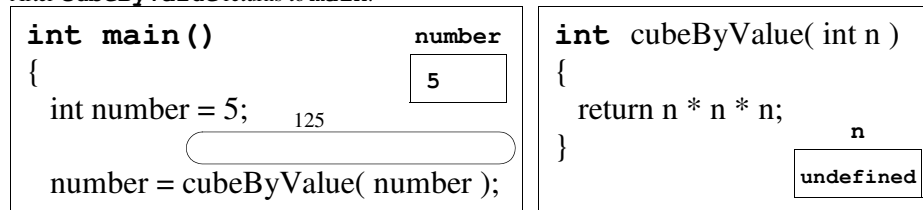
After **cubeByValue** receives the call:



After **cubeByValue** cubes the parameter **n**:



After **cubeByValue** returns to **main**:



After **main** completes the assignment to **number**:

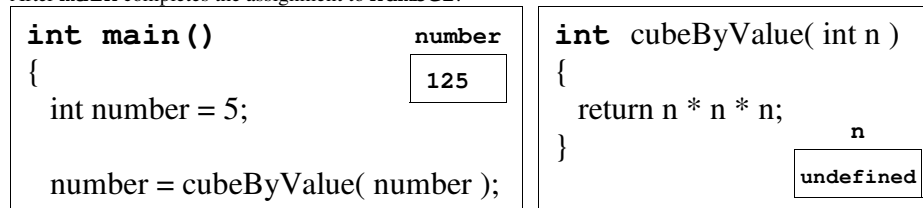


Fig. 5.8 Analysis of a typical call-by-value.

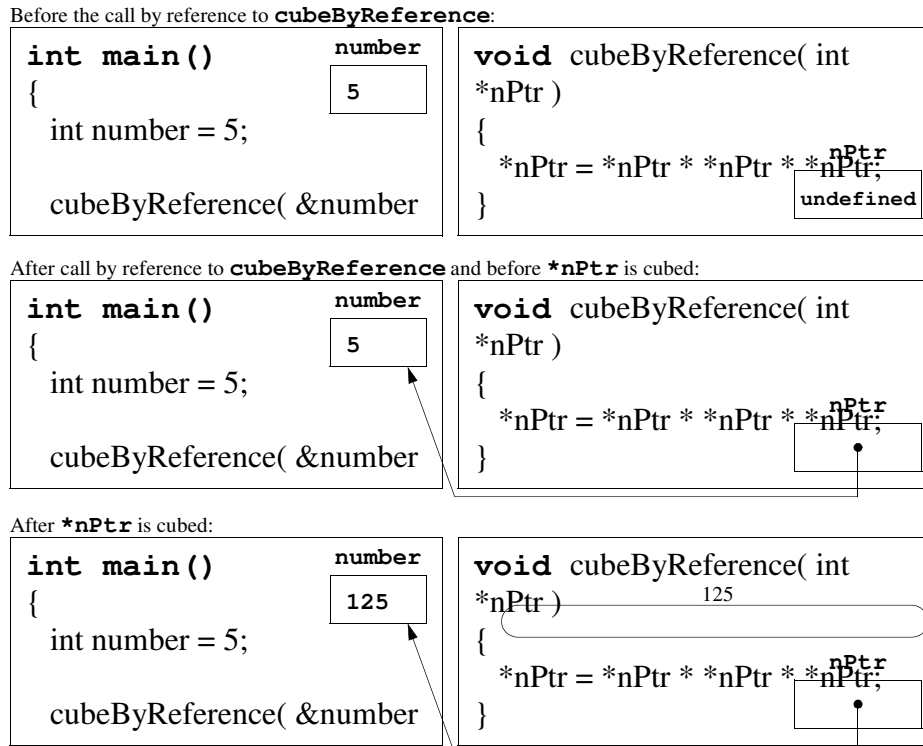


Fig. 5.9 Analysis of a typical call-by-reference with a pointer argument.

```

1 // Fig. 5.10: fig05_10.cpp
2 // Converting lowercase letters to uppercase letters
3 // using a non-constant pointer to non-constant data
4 #include <iostream.h>
5 #include <ctype.h>
6
7 void convertToUpperCase( char * );
8
9 int main()
10 {
11     char string[] = "characters and $32.98";
12
13     cout << "The string before conversion is: " << string;
14     convertToUpperCase( string );
15     cout << "\nThe string after conversion is: "
16         << string << endl;
17     return 0;
18 }
19
20 void convertToUpperCase( char *sPtr )
21 {
22     while ( *sPtr != '\0' ) {
23
24         if ( *sPtr >= 'a' && *sPtr <= 'z' )
25             *sPtr = toupper( *sPtr ); // convert to uppercase
26
27         ++sPtr; // move sPtr to the next character
28     }
29 }

```

```
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.

```
1 // Fig. 5.11: fig05_11.cpp
2 // Printing a string one character at a time using
3 // a non-constant pointer to constant data
4 #include <iostream.h>
5
6 void printCharacters( const char * );
7
8 int main()
9 {
10     char string[] = "print characters of a string";
11
12     cout << "The string is:\n";
13     printCharacters( string );
14     cout << endl;
15     return 0;
16 }
17
18 // In printCharacters, sPtr is a pointer to a character
19 // constant. Characters cannot be modified through sPtr
20 // (i.e., sPtr is a "read-only" pointer).
21 void printCharacters( const char *sPtr )
22 {
23     for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
24         cout << *sPtr;
25 }
```

```
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.

```

1 // Fig. 5.12: fig05_12.cpp
2 // Attempting to modify data through a
3 // non-constant pointer to constant data.
4 #include <iostream.h>
5
6 void f( const int * );
7
8 int main()
9 {
10     int y;

```

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

```

11
12     f( &y );    // f attempts illegal modification
13
14     return 0;
15 }
16
17 // In f, xPtr is a pointer to an integer constant
18 void f( const int *xPtr )
19 {
20     *xPtr = 100; // cannot modify a const object
21 }

```

```

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).

```

1 // Fig. 5.13: fig05_13.cpp
2 // Attempting to modify a constant pointer to
3 // non-constant data
4 #include <iostream.h>
5
6 int main()
7 {
8     int x, y;
9
10     int * const ptr = &x; // ptr is a constant pointer to an
11                          // integer. An integer can be modified
12                          // through ptr, but ptr always points
13                          // to the same memory location.
14     *ptr = 7;
15     ptr = &y;
16
17     return 0;
18 }

```

```

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
16    cout << *ptr << endl;
17    *ptr = 7;
18    ptr = &y;
19
20    return 0;
21 }

```

```

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.

```

1 // Fig. 5.15: fig05_15.cpp
2 // This program puts values into an array, sorts the values into
3 // ascending order, and prints the resulting array.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void bubbleSort( int *, const int );
8
9 int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13     int i;
14
15     cout << "Data items in original order\n";
16
17     for ( i = 0; i < arraySize; i++ )
18         cout << setw( 4 ) << a[ i ];
19
20     bubbleSort( a, arraySize ); // sort the array
21     cout << "\nData items in ascending order\n";
22
23     for ( i = 0; i < arraySize; i++ )
24         cout << setw( 4 ) << a[ i ];
25
26     cout << endl;
27     return 0;
28 }
29

```

```

30 void bubbleSort( int *array, const int size )
31 {
32     void swap( int *, int * );
33
34     for ( int pass = 0; pass < size - 1; pass++ )
35
36         for ( int j = 0; j < size - 1; j++ )
37
38             if ( array[ j ] > array[ j + 1 ] )
39                 swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44     int hold = *element1Ptr;
45     *element1Ptr = *element2Ptr;
46     *element2Ptr = hold;
47 }

```

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;

```

<pre> 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 </pre>

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

```

1 // Fig. 5.16: fig05_16.cpp
2 // Sizeof operator when used on an array name
3 // returns the number of bytes in the array.
4 #include <iostream.h>
5
6 size_t getSize( float * );
7
8 int main()
9 {
10     float array[ 20 ];
11
12     cout << "The number of bytes in the array is "
13          << sizeof( array )
14          << "\nThe number of bytes returned by getSize is "
15          << getSize( array ) << endl;
16
17     return 0;
18 }
19
20 size_t getSize( float *ptr )
21 {
22     return sizeof( ptr );
23 }

```

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.

```

1 // Fig. 5.17: fig05_17.cpp
2 // Demonstrating the sizeof operator
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     char c;
9     short s;
10    int i;
11    long l;
12    float f;
13    double d;
14    long double ld;
15    int array[ 20 ], *ptr = array;
16
17    cout << "sizeof c = " << sizeof c
18         << "\tsizeof(char) = " << sizeof( char )
19         << "\nsizeof s = " << sizeof s
20         << "\tsizeof(short) = " << sizeof( short )
21         << "\nsizeof i = " << sizeof i
22         << "\tsizeof(int) = " << sizeof( int )
23         << "\nsizeof l = " << sizeof l
24         << "\tsizeof(long) = " << sizeof( long )
25         << "\nsizeof f = " << sizeof f
26         << "\tsizeof(float) = " << sizeof( float )

```

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

```

27         << "\nsizeof d = " << sizeof d
28         << "\tsizeof(double) = " << sizeof( double )
29         << "\nsizeof ld = " << sizeof ld
30         << "\tsizeof(long double) = " << sizeof( long double )
31         << "\nsizeof array = " << sizeof array
32         << "\nsizeof ptr = " << sizeof ptr
33         << endl;
34    return 0;
35 }

```

```

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).

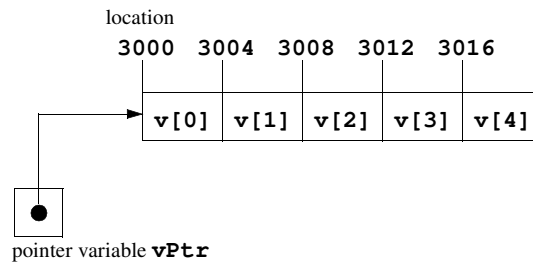


Fig. 5.18 The array `v` and a pointer variable `vPtr` that points to `v`.

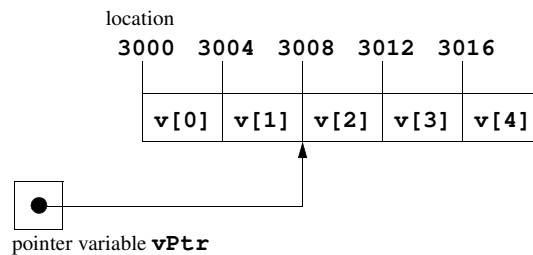


Fig. 5.19 The pointer `vPtr` after pointer arithmetic.

```
1 // Fig. 5.20: fig05_20.cpp
2 // Using subscripting and pointer notations with arrays
3
4 #include <iostream.h>
5
6 int main()
7 {
8     int b[] = { 10, 20, 30, 40 };
9     int *bPtr = b;    // set bPtr to point to array b
10
11     cout << "Array b printed with:\n"
12           << "Array subscript notation\n";
13
14     for ( int i = 0; i < 4; i++ )
15         cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17
18     cout << "\nPointer/offset notation where\n"
19           << "the pointer is the array name\n";
20
21     for ( int offset = 0; offset < 4; offset++ )
22         cout << "*(b + " << offset << ") = "
23               << *( b + offset ) << '\n';
24
25
26     cout << "\nPointer subscript notation\n";
27
28     for ( i = 0; i < 4; i++ )
29         cout << "bPtr[" << i << "] = " << bPtr[ i ] << '\n';
30
31     cout << "\nPointer/offset notation\n";
32
33     for ( offset = 0; offset < 4; offset++ )
34         cout << "*(bPtr + " << offset << ") = "
35               << *( bPtr + offset ) << '\n';
36
37     return 0;
38 }
```

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).

```

1 // Fig. 5.21: fig05_21.cpp
2 // Copying a string using array notation
3 // and pointer notation.
4 #include <iostream.h>
5
6 void copy1( char *, const char * );
7 void copy2( char *, const char * );
8
9 int main()
10 {
11     char string1[ 10 ], *string2 = "Hello",
12         string3[ 10 ], string4[] = "Good Bye";
13
14     copy1( string1, string2 );
15     cout << "string1 = " << string1 << endl;
16
17     copy2( string3, string4 );
18     cout << "string3 = " << string3 << endl;
19
20     return 0;
21 }

```

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

```

22
23 // copy s2 to s1 using array notation
24 void copy1( char *s1, const char *s2 )
25 {
26     for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
27         ; // do nothing in body
28 }
29
30 // copy s2 to s1 using pointer notation
31 void copy2( char *s1, const char *s2 )
32 {
33     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
34         ; // do nothing in body
35 }

```

```

string1 = Hello
string3 = Good Bye

```

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

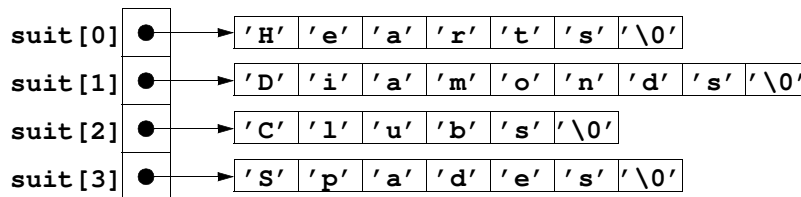


Fig. 5.22 A graphical representation of the `suit` array.

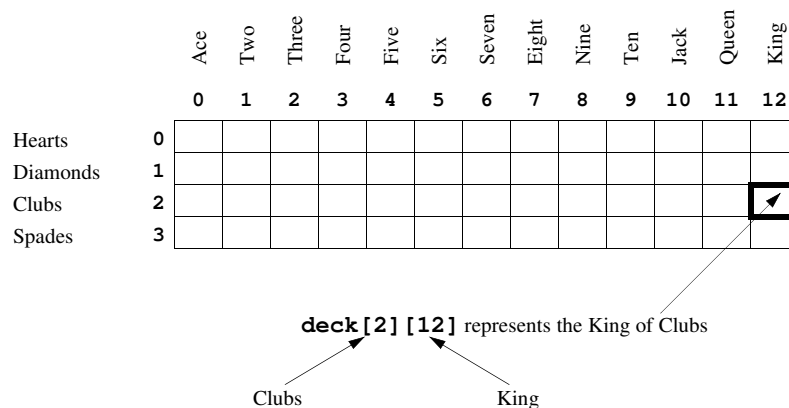


Fig. 5.23 Double-subscripted array representation of a deck of cards.

```

1 // Fig. 5.24: fig05_24.cpp
2 // Card shuffling and dealing program
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 void shuffle( int [][] [ 13 ] );
9 void deal( const int [][] [ 13 ], const char *[], const char *[] );
10
11 int main()
12 {
13     const char *suit[ 4 ] =
14         { "Hearts", "Diamonds", "Clubs", "Spades" };
15     const char *face[ 13 ] =
16         { "Ace", "Deuce", "Three", "Four",
17           "Five", "Six", "Seven", "Eight",
18           "Nine", "Ten", "Jack", "Queen", "King" };
19     int deck[ 4 ][ 13 ] = { 0 };
20
21     srand( time( 0 ) );
22
23     shuffle( deck );
24     deal( deck, face, suit );
25
26     return 0;
27 }
28
29 void shuffle( int wDeck[][] [ 13 ] )
30 {
31     int row, column;
32
33     for ( int card = 1; card <= 52; card++ ) {
34         do {
35             row = rand() % 4;
36             column = rand() % 13;
37         } while( wDeck[ row ][ column ] != 0 );
38
39         wDeck[ row ][ column ] = card;
40     }
41 }
42
43 void deal( const int wDeck[][] [ 13 ], const char *wFace[],
44           const char *wSuit[] )
45 {
46     for ( int card = 1; card <= 52; card++ )
47
48         for ( int row = 0; row <= 3; row++ )
49
50             for ( int column = 0; column <= 12; column++ )
51
52                 if ( wDeck[ row ][ column ] == card )
53                     cout << setw( 5 ) << setiosflags( ios::right )
54                         << wFace[ column ] << " of "
55                         << setw( 8 ) << setiosflags( ios::left )
56                         << wSuit[ row ]
57                         << ( card % 2 == 0 ? '\n' : '\t' );
58 }

```

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.

```

1 // Fig. 5.26: fig05_26.cpp
2 // Multipurpose sorting program using function pointers
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void bubble( int [], const int, int (*)( int, int ) );
7 int ascending( int, int );
8 int descending( int, int );
9
10 int main()
11 {
12     const int arraySize = 10;
13     int order,
14         counter,
15         a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16
17     cout << "Enter 1 to sort in ascending order,\n"
18          << "Enter 2 to sort in descending order: ";
19     cin >> order;
20     cout << "\nData items in original order\n";
21
22     for ( counter = 0; counter < arraySize; counter++ )
23         cout << setw( 4 ) << a[ counter ];
24
25     if ( order == 1 ) {
26         bubble( a, arraySize, ascending );
27         cout << "\nData items in ascending order\n";
28     }

```

```

29     else {
30         bubble( a, arraySize, descending );
31         cout << "\nData items in descending order\n";
32     }
33
34     for ( counter = 0; counter < arraySize; counter++ )
35         cout << setw( 4 ) << a[ counter ];
36
37     cout << endl;
38     return 0;
39 }
40
41 void bubble( int work[], const int size,
42             int (*compare)( int, int ) )
43 {
44     void swap( int *, int * );
45
46     for ( int pass = 1; pass < size; pass++ )

```

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

```

47         for ( int count = 0; count < size - 1; count++ )
48             if ( (*compare)( work[ count ], work[ count + 1 ] ) )
49                 swap( &work[ count ], &work[ count + 1 ] );
50
51     }
52
53 void swap( int *element1Ptr, int *element2Ptr )
54 {
55     int temp;
56
57     temp = *element1Ptr;
58     *element1Ptr = *element2Ptr;
59     *element2Ptr = temp;
60 }
61
62 int ascending( int a, int b )
63 {
64     return b < a;    // swap if b is less than a
65 }
66
67 int descending( int a, int b )
68 {
69     return b > a;    // swap if b is greater than a
70 }
71

```

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.

```

1  // Fig. 5.28: fig05_28.cpp
2  // Demonstrating an array of pointers to functions
3  #include <iostream.h>
4  void function1( int );
5  void function2( int );
6  void function3( int );
7
8  int main()
9  {
10     void (*f[ 3 ])( int ) = { function1, function2, function3 };
11     int choice;
12
13     cout << "Enter a number between 0 and 2, 3 to end: ";
14     cin >> choice;
15
16     while ( choice >= 0 && choice < 3 ) {
17         (*f[ choice ])( choice );
18         cout << "Enter a number between 0 and 2, 3 to end: ";
19         cin >> choice;
20     }
21
22     cout << "Program execution completed." << endl;
23     return 0;
24 }
25
26 void function1( int a )
27 {
28     cout << "You entered " << a
29         << " so function1 was called\n\n";
30 }
31
32 void function2( int b )
33 {
34     cout << "You entered " << b
35         << " so function2 was called\n\n";
36 }
37
38 void function3( int c )
39 {
40     cout << "You entered " << c
41         << " so function3 was called\n\n";
42 }

```

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
<code>char *strcpy(char *s1, const char *s2)</code>	Copies the string <code>s2</code> into the character array <code>s1</code> . The value of <code>s1</code> is returned.
<code>char *strncpy(char *s1, const char *s2, size_t n)</code>	Copies at most <code>n</code> characters of the string <code>s2</code> into the character array <code>s1</code> . The value of <code>s1</code> is returned.
<code>char *strcat(char *s1, const char *s2)</code>	Appends the string <code>s2</code> to the string <code>s1</code> . The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code> . The value of <code>s1</code> is returned.
<code>char *strncat(char *s1, const char *s2, size_t n)</code>	Appends at most <code>n</code> characters of string <code>s2</code> to string <code>s1</code> . The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code> . The value of <code>s1</code> is returned.
<code>int strcmp(const char *s1, const char *s2)</code>	Compares the string <code>s1</code> to the string <code>s2</code> . The function returns a value of 0, less than 0, or greater than 0 if <code>s1</code> is equal to, less than, or greater than <code>s2</code> , respectively.
<code>int strncmp(const char *s1, const char *s2, size_t n)</code>	Compares up to <code>n</code> characters of the string <code>s1</code> to the string <code>s2</code> . The function returns 0, less than 0, or greater than 0 if <code>s1</code> is equal to, less than, or greater than <code>s2</code> , respectively.
<code>char *strtok(char *s1, const char *s2)</code>	A sequence of calls to <code>strtok</code> breaks string <code>s1</code> into “tokens”—logical pieces such as words in a line of text—separated by characters contained in string <code>s2</code> . The first call contains <code>s1</code> as the first argument, and subsequent calls to continue tokenizing the same string contain <code>NULL</code> 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, <code>NULL</code> is returned.
<code>size_t strlen(const char *s)</code>	Determines the length of string <code>s</code> . The number of characters preceding the terminating null character is returned.

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

```

1 // Fig. 5.30: fig05_30.cpp
2 // Using strcpy and strncpy
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char x[] = "Happy Birthday to You";
9     char y[ 25 ], z[ 15 ];
10
11     cout << "The string in array x is: " << x
12         << "\nThe string in array y is: " << strcpy( y, x )
13         << '\n';
14     strncpy( z, x, 14 ); // does not copy null character
15     z[ 14 ] = '\0';
16     cout << "The string in array z is: " << z << endl;
17
18     return 0;
19 }

```

```

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**.

```

1 // Fig. 5.31: fig05_31.cpp
2 // Using strcat and strncat
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char s1[ 20 ] = "Happy ";
9     char s2[] = "New Year ";
10    char s3[ 40 ] = "";
11
12    cout << "s1 = " << s1 << "\ns2 = " << s2;
13    cout << "\nstrcat(s1, s2) = " << strcat( s1, s2 );
14    cout << "\nstrncat(s3, s1, 6) = " << strncat( s3, s1, 6 );
15    cout << "\nstrcat(s3, s1) = " << strcat( s3, s1 ) << endl;
16
17    return 0;
18 }

```

```

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**.

```

1 // Fig. 5.32: fig05_32.cpp
2 // Using strcmp and strncmp
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <string.h>
6
7 int main()
8 {
9     char *s1 = "Happy New Year";
10    char *s2 = "Happy New Year";
11    char *s3 = "Happy Holidays";
12
13    cout << "s1 = " << s1 << "\ns2 = " << s2
14         << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
15         << setw( 2 ) << strcmp( s1, s2 )
16         << "\nstrcmp(s1, s3) = " << setw( 2 )
17         << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
18         << setw( 2 ) << strcmp( s3, s1 );
19
20    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
21         << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
22         << setw( 2 ) << strncmp( s1, s3, 7 )
23         << "\nstrncmp(s3, s1, 7) = "
24         << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
25    return 0;
26 }

```

```

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`.

```

1 // Fig. 5.33: fig05_33.cpp
2 // Using strtok
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char string[] = "This is a sentence with 7 tokens";
9     char *tokenPtr;
10
11    cout << "The string to be tokenized is:\n" << string
12         << "\n\nThe tokens are:\n";
13
14    tokenPtr = strtok( string, " " );
15
16    while ( tokenPtr != NULL ) {

```

```

17     cout << tokenPtr << '\n';
18     tokenPtr = strtok( NULL, " " );
19 }
20
21 return 0;
22 }

```

```

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`.

```

1 // Fig. 5.34: fig05_34.cpp
2 // Using strlen
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "abcdefghijklmnopqrstuvwxyz";
9     char *string2 = "four";
10    char *string3 = "Boston";
11
12    cout << "The length of \"" << string1
13         << "\" is " << strlen( string1 )
14         << "\n";
15    cout << "The length of \"" << string2
16         << "\" is " << strlen( string2 )
17         << "\n";
18    cout << "The length of \"" << string3
19         << "\" is " << strlen( string3 ) << endl;
20
21    return 0;
22 }

```

```

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** Creating a structure, setting its members, and printing the structure.
- Fig. 6.2** Simple definition of **class Time**.
- Fig. 6.3** Abstract data type **Time** implementation as a class.
- 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** Separating **Time** class interface and implementation.
- Fig. 6.6** Erroneous attempt to access **private** members of a class.
- Fig. 6.7** Using a utility function.
- Fig. 6.8** Using a constructor with default arguments.
- Fig. 6.9** Demonstrating the order in which constructors and destructors are called.
- Fig. 6.10** Using set and get functions.
- Fig. 6.11** Returning a reference to a private data member.


```

1 // Fig. 6.1: fig06_01.cpp
2 // Create a structure, set its members, and print it.
3 #include <iostream.h>
4
5 struct Time { // structure definition
6     int hour; // 0-23
7     int minute; // 0-59
8     int second; // 0-59
9 };
10
11 void printMilitary( const Time & ); // prototype
12 void printStandard( const Time & ); // prototype
13
14 int main()
15 {
16     Time dinnerTime; // variable of new type Time
17
18     // set members to valid values
19     dinnerTime.hour = 18;
20     dinnerTime.minute = 30;
21     dinnerTime.second = 0;

```

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

```

22
23     cout << "Dinner will be held at ";
24     printMilitary( dinnerTime );
25     cout << " military time,\nwhich is ";
26     printStandard( dinnerTime );
27     cout << " standard time.\n";
28
29     // set members to invalid values
30     dinnerTime.hour = 29;
31     dinnerTime.minute = 73;
32
33     cout << "\nTime with invalid values: ";
34     printMilitary( dinnerTime );
35     cout << endl;
36     return 0;
37 }
38
39 // Print the time in military format
40 void printMilitary( const Time &t )
41 {
42     cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
43         << ( t.minute < 10 ? "0" : "" ) << t.minute;
44 }
45
46 // Print the time in standard format
47 void printStandard( const Time &t )
48 {
49     cout << ( ( t.hour == 0 || t.hour == 12 ) ?
50         12 : t.hour % 12 )
51         << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
52         << ":" << ( t.second < 10 ? "0" : "" ) << t.second
53         << ( t.hour < 12 ? " AM" : " PM" );
54 }

```

```
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).

```
1 class Time {
2 public:
3     Time();
4     void setTime( int, int, int );
5     void printMilitary();
6     void printStandard();
7 private:
8     int hour;        // 0 - 23
9     int minute;     // 0 - 59
10    int second;     // 0 - 59
11 };
```

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

```
1 // Fig. 6.3: fig06_03.cpp
2 // Time class.
3 #include <iostream.h>
4
5 // Time abstract data type (ADT) definition
6 class Time {
7 public:
8     Time(); // constructor
9     void setTime( int, int, int ); // set hour, minute, second
10    void printMilitary(); // print military time format
11    void printStandard(); // print standard time format
12 private:
13    int hour; // 0 - 23
14    int minute; // 0 - 59
15    int second; // 0 - 59
16 };
17
18 // Time constructor initializes each data member to zero.
19 // Ensures all Time objects start in a consistent state.
20 Time::Time() { hour = minute = second = 0; }
```

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

```
21
22 // Set a new Time value using military time. Perform validity
23 // checks on the data values. Set invalid values to zero.
24 void Time::setTime( int h, int m, int s )
25 {
26     hour = ( h >= 0 && h < 24 ) ? h : 0;
27     minute = ( m >= 0 && m < 60 ) ? m : 0;
28     second = ( s >= 0 && s < 60 ) ? s : 0;
29 }
30
31 // Print Time in military format
```

```

32 void Time::printMilitary()
33 {
34     cout << ( ( hour < 10 ? "0" : "" ) << hour << ":"
35             << ( minute < 10 ? "0" : "" ) << minute;
36 }
37
38 // Print Time in standard format
39 void Time::printStandard()
40 {
41     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
42             << ":" << ( minute < 10 ? "0" : "" ) << minute
43             << ":" << ( second < 10 ? "0" : "" ) << second
44             << ( hour < 12 ? " AM" : " PM" );
45 }
46
47 // Driver to test simple class Time
48 int main()
49 {
50     Time t; // instantiate object t of class Time
51
52     cout << "The initial military time is ";
53     t.printMilitary();
54     cout << "\nThe initial standard time is ";
55     t.printStandard();
56
57     t.setTime( 13, 27, 6 );
58     cout << "\n\nMilitary time after setTime is ";
59     t.printMilitary();
60     cout << "\nStandard time after setTime is ";
61     t.printStandard();
62
63     t.setTime( 99, 99, 99 ); // attempt invalid settings
64     cout << "\n\nAfter attempting invalid settings:"
65             << "\nMilitary time: ";
66     t.printMilitary();
67     cout << "\nStandard time: ";
68     t.printStandard();
69     cout << endl;
70     return 0;
71 }

```

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).

```
1 // Fig. 6.4: fig06_04.cpp
2 // Demonstrating the class member access operators . and ->
3 //
4 // CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
5 #include <iostream.h>
6
7 // Simple class Count
8 class Count {
9 public:
10     int x;
11     void print() { cout << x << endl; }
12 };
13
14 int main()
15 {
16     Count counter,           // create counter object
17         *counterPtr = &counter, // pointer to counter
18         &counterRef = counter; // reference to counter
19
20     cout << "Assign 7 to x and print using the object's name: ";
21     counter.x = 7;           // assign 7 to data member x
22     counter.print();        // call member function print
23
24     cout << "Assign 8 to x and print using a reference: ";
25     counterRef.x = 8;       // assign 8 to data member x
26     counterRef.print();     // call member function print
27
28     cout << "Assign 10 to x and print using a pointer: ";
29     counterPtr->x = 10;     // assign 10 to data member x
30     counterPtr->print();    // call member function print
31     return 0;
32 }
```

```
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.

```

1 // Fig. 6.5: time1.h
2 // Declaration of the Time class.
3 // Member functions are defined in time1.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME1_H
7 #define TIME1_H
8
9 // Time abstract data type definition
10 class Time {
11 public:
12     Time(); // constructor
13     void setTime( int, int, int ); // set hour, minute, second
14     void printMilitary(); // print military time format
15     void printStandard(); // print standard time format
16 private:
17     int hour; // 0 - 23
18     int minute; // 0 - 59
19     int second; // 0 - 59
20 };
21
22 #endif

```

Fig. 6.5 Separating **Time** class interface and implementation (part 1 of 5).

```

23 // Fig. 6.5: time1.cpp
24 // Member function definitions for Time class.
25 #include <iostream.h>
26 #include "time1.h"
27
28 // Time constructor initializes each data member to zero.
29 // Ensures all Time objects start in a consistent state.
30 Time::Time() { hour = minute = second = 0; }
31
32 // Set a new Time value using military time. Perform validity
33 // checks on the data values. Set invalid values to zero.
34 void Time::setTime( int h, int m, int s )
35 {
36     hour = ( h >= 0 && h < 24 ) ? h : 0;
37     minute = ( m >= 0 && m < 60 ) ? m : 0;
38     second = ( s >= 0 && s < 60 ) ? s : 0;
39 }
40
41 // Print Time in military format
42 void Time::printMilitary()
43 {
44     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
45           << ( minute < 10 ? "0" : "" ) << minute;
46 }
47

```

Fig. 6.5 Separating **Time** class interface and implementation (part 2 of 5).

```

48 // Print time in standard format
49 void Time::printStandard()
50 {
51     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
52           << ":" << ( minute < 10 ? "0" : "" ) << minute
53           << ":" << ( second < 10 ? "0" : "" ) << second
54           << ( hour < 12 ? " AM" : " PM" );
55 }

```

Fig. 6.5 Separating **Time** class interface and implementation (part 3 of 5).

```
56 // Fig. 6.5: fig06_05.cpp
57 // Driver for Time1 class
58 // NOTE: Compile with time1.cpp
59 #include <iostream.h>
60 #include "time1.h"
61
62 // Driver to test simple class Time
63 int main()
64 {
65     Time t; // instantiate object t of class time
66
67     cout << "The initial military time is ";
68     t.printMilitary();
69     cout << "\nThe initial standard time is ";
70     t.printStandard();
71
72     t.setTime( 13, 27, 6 );
73     cout << "\n\nMilitary time after setTime is ";
74     t.printMilitary();
75     cout << "\nStandard time after setTime is ";
76     t.printStandard();
77
78     t.setTime( 99, 99, 99 ); // attempt invalid settings
79     cout << "\n\nAfter attempting invalid settings:\n"
80         << "Military time: ";
81     t.printMilitary();
82     cout << "\nStandard time: ";
83     t.printStandard();
84     cout << endl;
85     return 0;
86 }
```

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).

```

1 // Fig. 6.6: fig06_06.cpp
2 // Demonstrate errors resulting from attempts
3 // to access private class members.
4 #include <iostream.h>
5 #include "time1.h"
6
7 int main()
8 {
9     Time t;
10
11     // Error: 'Time::hour' is not accessible
12     t.hour = 7;
13
14     // Error: 'Time::minute' is not accessible
15     cout << "minute = " << t.minute;
16
17     return 0;
18 }

```

```

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.

```

1 // Fig. 6.7: salesp.h
2 // SalesPerson class definition
3 // Member functions defined in salesp.cpp
4 #ifndef SALESP_H
5 #define SALESP_H
6
7 class SalesPerson {
8 public:
9     SalesPerson(); // constructor
10    void getSalesFromUser(); // get sales figures from keyboard
11    void setSales( int, double ); // User supplies one month's
12                                   // sales figures.
13    void printAnnualSales();
14
15 private:
16    double totalAnnualSales(); // utility function
17    double sales[ 12 ]; // 12 monthly sales figures
18 };
19
20 #endif

```

Fig. 6.7 Using a utility function (part 1 of 5).

```

21 // Fig. 6.7: salesp.cpp
22 // Member functions for class SalesPerson
23 #include <iostream.h>
24 #include <iomanip.h>
25 #include "salesp.h"
26
27 // Constructor function initializes array
28 SalesPerson::SalesPerson()
29 {
30     for ( int i = 0; i < 12; i++ )
31         sales[ i ] = 0.0;
32 }
33
34 // Function to get 12 sales figures from the user
35 // at the keyboard
36 void SalesPerson::getSalesFromUser()
37 {
38     double salesFigure;
39
40     for ( int i = 0; i < 12; i++ ) {
41         cout << "Enter sales amount for month "
42             << i + 1 << ": ";
43         cin >> salesFigure;
44         setSales( i, salesFigure );
45     }
46 }
47
48 // Function to set one of the 12 monthly sales figures.
49 // Note that the month value must be from 0 to 11.
50 void SalesPerson::setSales( int month, double amount )
51 {
52     if ( month >= 0 && month < 12 && amount > 0 )
53         sales[ month ] = amount;
54     else
55         cout << "Invalid month or sales figure" << endl;
56 }
57

```

Fig. 6.7 Using a utility function (part 2 of 5).

```

58 // Print the total annual sales
59 void SalesPerson::printAnnualSales()
60 {
61     cout << setprecision( 2 )
62         << setiosflags( ios::fixed | ios::showpoint )
63         << "\nThe total annual sales are: $"
64         << totalAnnualSales() << endl;
65 }
66
67 // Private utility function to total annual sales
68 double SalesPerson::totalAnnualSales()
69 {
70     double total = 0.0;
71
72     for ( int i = 0; i < 12; i++ )
73         total += sales[ i ];
74
75     return total;
76 }

```

Fig. 6.7 Using a utility function (part 3 of 5).


```

77 // Fig. 6.7: fig06_07.cpp
78 // Demonstrating a utility function
79 // Compile with salesp.cpp
80 #include "salesp.h"
81
82 int main()
83 {
84     SalesPerson s;          // create SalesPerson object s
85
86     s.getSalesFromUser(); // note simple sequential code
87     s.printAnnualSales(); // no control structures in main
88     return 0;
89 }

```

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).

```

1 // Fig. 6.8: time2.h
2 // Declaration of the Time class.
3 // Member functions are defined in time2.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 #ifndef TIME2_H
8 #define TIME2_H
9
10 // Time abstract data type definition
11 class Time {
12 public:
13     Time( int = 0, int = 0, int = 0 ); // default constructor
14     void setTime( int, int, int ); // set hour, minute, second
15     void printMilitary(); // print military time format
16     void printStandard(); // print standard time format
17 private:
18     int hour; // 0 - 23
19     int minute; // 0 - 59
20     int second; // 0 - 59
21 };
22
23 #endif

```

Fig. 6.8 Using a constructor with default arguments (part 1 of 6).

```

24 // Fig. 6.8: time2.cpp
25 // Member function definitions for Time class.
26 #include <iostream.h>
27 #include "time2.h"
28
29 // Time constructor initializes each data member to zero.
30 // Ensures all Time objects start in a consistent state.
31 Time::Time( int hr, int min, int sec )
32     { setTime( hr, min, sec ); }
33

```

Fig. 6.8 Using a constructor with default arguments (part 2 of 6).

```

34 // Set a new Time value using military time. Perform validity
35 // checks on the data values. Set invalid values to zero.
36 void Time::setTime( int h, int m, int s )
37 {
38     hour   = ( h >= 0 && h < 24 ) ? h : 0;
39     minute = ( m >= 0 && m < 60 ) ? m : 0;
40     second = ( s >= 0 && s < 60 ) ? s : 0;
41 }
42
43 // Print Time in military format
44 void Time::printMilitary()
45 {
46     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
47           << ( minute < 10 ? "0" : "" ) << minute;
48 }
49
50 // Print Time in standard format
51 void Time::printStandard()
52 {
53     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
54           << ":" << ( minute < 10 ? "0" : "" ) << minute
55           << ":" << ( second < 10 ? "0" : "" ) << second
56           << ( hour < 12 ? " AM" : " PM" );
57 }

```

Fig. 6.8 Using a constructor with default arguments (part 3 of 6).

```

58 // Fig. 6.8: fig06_08.cpp
59 // Demonstrating a default constructor
60 // function for class Time.
61 #include <iostream.h>
62 #include "time2.h"
63
64 int main()
65 {
66     Time t1,           // all arguments defaulted
67         t2(2),        // minute and second defaulted
68         t3(21, 34),   // second defaulted
69         t4(12, 25, 42), // all values specified
70         t5(27, 74, 99); // all bad values specified
71
72     cout << "Constructed with:\n"
73           << "all arguments defaulted:\n  ";
74     t1.printMilitary();
75     cout << "\n  ";
76     t1.printStandard();
77
78     cout << "\nhour specified; minute and second defaulted:"

```

```
79         << "\n ";
```

Fig. 6.8 Using a constructor with default arguments (part 4 of 6).

```
80     t2.printMilitary();
81     cout << "\n ";
82     t2.printStandard();
83
84     cout << "\nhour and minute specified; second defaulted:"
85           << "\n ";
86     t3.printMilitary();
87     cout << "\n ";
88     t3.printStandard();
89
90     cout << "\nhour, minute, and second specified:"
91           << "\n ";
92     t4.printMilitary();
93     cout << "\n ";
94     t4.printStandard();
95
96     cout << "\nall invalid values specified:"
97           << "\n ";
98     t5.printMilitary();
99     cout << "\n ";
100    t5.printStandard();
101    cout << endl;
102
103    return 0;
104 }
```

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).

```

1 // Fig. 6.9: create.h
2 // Definition of class CreateAndDestroy.
3 // Member functions defined in create.cpp.
4 #ifndef CREATE_H
5 #define CREATE_H
6
7 class CreateAndDestroy {
8 public:
9     CreateAndDestroy( int ); // constructor
10    ~CreateAndDestroy();     // destructor
11 private:
12    int data;
13 };
14
15 #endif

```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 1 of 4).

```

16 // Fig. 6.9: create.cpp
17 // Member function definitions for class CreateAndDestroy
18 #include <iostream.h>
19 #include "create.h"
20
21 CreateAndDestroy::CreateAndDestroy( int value )
22 {
23     data = value;
24     cout << "Object " << data << " constructor";
25 }
26
27 CreateAndDestroy::~~CreateAndDestroy()
28 { cout << "Object " << data << " destructor " << endl; }

```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 2 of 4).

```

29 // Fig. 6.9: fig06_09.cpp
30 // Demonstrating the order in which constructors and
31 // destructors are called.
32 #include <iostream.h>
33 #include "create.h"
34
35 void create( void ); // prototype
36
37 CreateAndDestroy first( 1 ); // global object
38
39 int main()
40 {
41     cout << " (global created before main)" << endl;

```

Fig. 6.9 Demonstrating the order in which constructors and destructors are called (part 3 of 4).

```

42
43     CreateAndDestroy second( 2 ); // local object
44     cout << " (local automatic in main)" << endl;
45
46     static CreateAndDestroy third( 3 ); // local object
47     cout << " (local static in main)" << endl;
48
49     create(); // call function to create objects
50
51     CreateAndDestroy fourth( 4 ); // local object
52     cout << " (local automatic in main)" << endl;
53     return 0;
54 }

```

```

55
56 // Function to create objects
57 void create( void )
58 {
59     CreateAndDestroy fifth( 5 );
60     cout << "    (local automatic in create)" << endl;
61
62     static CreateAndDestroy sixth( 6 );
63     cout << "    (local static in create)" << endl;
64
65     CreateAndDestroy seventh( 7 );
66     cout << "    (local automatic in create)" << endl;
67 }

```

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).

```

1 // Fig. 6.10: time3.h
2 // Declaration of the Time class.
3 // Member functions defined in time3.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 #ifndef TIME3_H
8 #define TIME3_H
9
10 class Time {
11 public:
12     Time( int = 0, int = 0, int = 0 ); // constructor
13
14     // set functions
15     void setTime( int, int, int ); // set hour, minute, second
16     void setHour( int ); // set hour
17     void setMinute( int ); // set minute
18     void setSecond( int ); // set second

```

Fig. 6.10 Using set and get functions (part 1 of 6).

```

19
20     // get functions
21     int getHour();           // return hour
22     int getMinute();       // return minute
23     int getSecond();       // return second
24
25     void printMilitary();   // output military time
26     void printStandard();  // output standard time
27
28 private:
29     int hour;               // 0 - 23
30     int minute;            // 0 - 59
31     int second;            // 0 - 59
32 };
33
34 #endif

```

Fig. 6.10 Using set and get functions (part 2 of 6).

```

35 // Fig. 6.10: time3.cpp
36 // Member function definitions for Time class.
37 #include "time3.h"
38 #include <iostream.h>
39
40 // Constructor function to initialize private data.
41 // Calls member function setTime to set variables.
42 // Default values are 0 (see class definition).
43 Time::Time( int hr, int min, int sec )
44     { setTime( hr, min, sec ); }
45
46 // Set the values of hour, minute, and second.
47 void Time::setTime( int h, int m, int s )
48 {
49     setHour( h );
50     setMinute( m );
51     setSecond( s );
52 }
53
54 // Set the hour value
55 void Time::setHour( int h )
56     { hour = ( h >= 0 && h < 24 ) ? h : 0; }
57
58 // Set the minute value
59 void Time::setMinute( int m )
60     { minute = ( m >= 0 && m < 60 ) ? m : 0; }
61
62 // Set the second value
63 void Time::setSecond( int s )
64     { second = ( s >= 0 && s < 60 ) ? s : 0; }
65

```

Fig. 6.10 Using set and get functions (part 3 of 6).

```

66 // Get the hour value
67 int Time::getHour() { return hour; }
68
69 // Get the minute value
70 int Time::getMinute() { return minute; }
71
72 // Get the second value
73 int Time::getSecond() { return second; }
74
75 // Print time in military format

```

```

76 void Time::printMilitary()
77 {
78     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
79         << ( minute < 10 ? "0" : "" ) << minute;
80 }
81
82 // Print time in standard format
83 void Time::printStandard()
84 {
85     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
86         << ":" << ( minute < 10 ? "0" : "" ) << minute
87         << ":" << ( second < 10 ? "0" : "" ) << second
88         << ( hour < 12 ? " AM" : " PM" );
89 }

```

Fig. 6.10 Using set and get functions (part 4 of 6).

```

90 // Fig. 6.10: fig06_10.cpp
91 // Demonstrating the Time class set and get functions
92 #include <iostream.h>
93 #include "time3.h"
94
95 void incrementMinutes( Time &, const int );
96
97 int main()
98 {
99     Time t;
100
101     t.setHour( 17 );
102     t.setMinute( 34 );
103     t.setSecond( 25 );
104
105     cout << "Result of setting all valid values:\n"
106         << " Hour: " << t.getHour()
107         << " Minute: " << t.getMinute()
108         << " Second: " << t.getSecond();
109
110     t.setHour( 234 ); // invalid hour set to 0
111     t.setMinute( 43 );
112     t.setSecond( 6373 ); // invalid second set to 0

```

Fig. 6.10 Using set and get functions (part 5 of 6).

```

113
114     cout << "\n\nResult of attempting to set invalid hour and"
115         << " second:\n Hour: " << t.getHour()
116         << " Minute: " << t.getMinute()
117         << " Second: " << t.getSecond() << "\n\n";
118
119     t.setTime( 11, 58, 0 );
120     incrementMinutes( t, 3 );
121
122     return 0;
123 }
124
125 void incrementMinutes(Time &tt, const int count)
126 {
127     cout << "Incrementing minute " << count
128         << " times:\nStart time: ";
129     tt.printStandard();
130
131     for ( int i = 0; i < count; i++ ) {
132         tt.setMinute( ( tt.getMinute() + 1 ) % 60);

```

```

133
134     if ( tt.getMinute() == 0 )
135         tt.setHour( ( tt.getHour() + 1 ) % 24);
136
137     cout << "\nminute + 1: ";
138     tt.printStandard();
139 }
140
141 cout << endl;
142 }

```

```

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).

```

1 // Fig. 6.11: time4.h
2 // Declaration of the Time class.
3 // Member functions defined in time4.cpp
4
5 // preprocessor directives that
6 // prevent multiple inclusions of header file
7 #ifndef TIME4_H
8 #define TIME4_H
9
10 class Time {
11 public:
12     Time( int = 0, int = 0, int = 0 );
13     void setTime( int, int, int );
14     int getHour();
15     int &badSetHour( int ); // DANGEROUS reference return
16 private:
17     int hour;
18     int minute;
19     int second;
20 };
21
22 #endif

```

Fig. 6.11 Returning a reference to a private data member (part 1 of 4).


```

23 // Fig. 6.11: time4.cpp
24 // Member function definitions for Time class.
25 #include "time4.h"
26 #include <iostream.h>
27
28 // Constructor function to initialize private data.
29 // Calls member function setTime to set variables.
30 // Default values are 0 (see class definition).
31 Time::Time( int hr, int min, int sec )
32     { setTime( hr, min, sec ); }
33
34 // Set the values of hour, minute, and second.
35 void Time::setTime( int h, int m, int s )
36     {
37     hour   = ( h >= 0 && h < 24 ) ? h : 0;
38     minute = ( m >= 0 && m < 60 ) ? m : 0;
39     second = ( s >= 0 && s < 60 ) ? s : 0;
40     }
41
42 // Get the hour value
43 int Time::getHour() { return hour; }
44
45 // POOR PROGRAMMING PRACTICE:
46 // Returning a reference to a private data member.
47 int &Time::badSetHour( int hh )
48     {
49     hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
50
51     return hour; // DANGEROUS reference return
52     }

```

Fig. 6.11 Returning a reference to a private data member (part 2 of 4).

```

53 // Fig. 6.11: fig06_11.cpp
54 // Demonstrating a public member function that
55 // returns a reference to a private data member.
56 // Time class has been trimmed for this example.
57 #include <iostream.h>
58 #include "time4.h"
59
60 int main()
61     {
62     Time t;
63     int &hourRef = t.badSetHour( 20 );
64
65     cout << "Hour before modification: " << hourRef;
66     hourRef = 30; // modification with invalid value
67     cout << "\nHour after modification: " << t.getHour();
68

```

Fig. 6.11 Returning a reference to a private data member (part 3 of 4).

```

69 // Dangerous: Function call that returns
70 // a reference can be used as an lvalue!
71 t.badSetHour(12) = 74;
72 cout << "\n\n*****\n"
73     << "POOR PROGRAMMING PRACTICE!!!!!!\n"
74     << "badSetHour as an lvalue, Hour: "
75     << t.getHour()
76     << "\n*****" << endl;
77
78 return 0;
79 }

```

```
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** Using a **Time** class with **const** objects and **const** member functions.
- Fig. 7.2** Using a member initializer to initialize a constant of a built-in data type.
- Fig. 7.3** Erroneous attempt to initialize a constant of a built-in data type by assignment.
- Fig. 7.4** Using member-object initializers.
- Fig. 7.5** Friends can access **private** members of a class.
- Fig. 7.6** Non-**friend**/non-member functions cannot access **private** class members.
- Fig. 7.7** Using the **this** pointer.
- Fig. 7.8** Cascading member function calls.
- Fig. 7.9** Using a **static** data member to maintain a count of the number of objects of a class.
- Fig. 7.10** Implementing a proxy class.

```

1 // Fig. 7.1: time5.h
2 // Declaration of the class Time.
3 // Member functions defined in time5.cpp
4 #ifndef TIME5_H
5 #define TIME5_H
6
7 class Time {
8 public:
9     Time( int = 0, int = 0, int = 0 ); // default constructor
10
11     // set functions
12     void setTime( int, int, int ); // set time
13     void setHour( int ); // set hour
14     void setMinute( int ); // set minute
15     void setSecond( int ); // set second
16
17     // get functions (normally declared const)
18     int getHour() const; // return hour
19     int getMinute() const; // return minute
20     int getSecond() const; // return second
21
22     // print functions (normally declared const)
23     void printMilitary() const; // print military time
24     void printStandard(); // print standard time

```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 1 of 6).

```

25 private:
26     int hour; // 0 - 23
27     int minute; // 0 - 59
28     int second; // 0 - 59
29 };
30
31 #endif

```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 2 of 6).

```

32 // Fig. 7.1: time5.cpp
33 // Member function definitions for Time class.
34 #include <iostream.h>
35 #include "time5.h"
36
37 // Constructor function to initialize private data.
38 // Default values are 0 (see class definition).
39 Time::Time( int hr, int min, int sec )
40     { setTime( hr, min, sec ); }
41
42 // Set the values of hour, minute, and second.
43 void Time::setTime( int h, int m, int s )
44     {
45         setHour( h );
46         setMinute( m );
47         setSecond( s );
48     }
49
50 // Set the hour value
51 void Time::setHour( int h )
52     { hour = ( h >= 0 && h < 24 ) ? h : 0; }
53
54 // Set the minute value
55 void Time::setMinute( int m )
56     { minute = ( m >= 0 && m < 60 ) ? m : 0; }
57

```

```

58 // Set the second value
59 void Time::setSecond( int s )
60     { second = ( s >= 0 && s < 60 ) ? s : 0; }
61
62 // Get the hour value
63 int Time::getHour() const { return hour; }
64
65 // Get the minute value
66 int Time::getMinute() const { return minute; }
67
68 // Get the second value
69 int Time::getSecond() const { return second; }

```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 3 of 6).

```

70
71 // Display military format time: HH:MM
72 void Time::printMilitary() const
73 {
74     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
75         << ( minute < 10 ? "0" : "" ) << minute;
76 }
77
78 // Display standard format time: HH:MM:SS AM (or PM)
79 void Time::printStandard()
80 {
81     cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
82         << ( minute < 10 ? "0" : "" ) << minute << ":"
83         << ( second < 10 ? "0" : "" ) << second
84         << ( hour < 12 ? " AM" : " PM" );
85 }

```

Fig. 7.1 Using a `Time` class with `const` objects and `const` member functions (part 4 of 6).

```

86 // Fig. 7.1: fig07_01.cpp
87 // Attempting to access a const object with
88 // non-const member functions.
89 #include <iostream.h>
90 #include "time5.h"
91
92 int main()
93 {
94     Time wakeUp( 6, 45, 0 ); // non-constant object
95     const Time noon( 12, 0, 0 ); // constant object
96
97     // MEMBER FUNCTION    OBJECT
98     wakeUp.setHour( 18 ); // non-const    non-const
99
100    noon.setHour( 12 ); // non-const    const
101
102    wakeUp.getHour(); // const    non-const
103
104    noon.getMinute(); // const    const
105    noon.printMilitary(); // const    const
106    noon.printStandard(); // non-const    const
107    return 0;
108 }

```

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).

```

1 // Fig. 7.2: fig07_02.cpp
2 // Using a member initializer to initialize a
3 // constant of a built-in data type.
4
5 #include <iostream.h>
6
7 class Increment {
8 public:
9     Increment( int c = 0, int i = 1 );
10    void addIncrement() { count += increment; }
11    void print() const;
12
13 private:
14    int count;
15    const int increment; // const data member
16 };
17
18 // Constructor for class Increment
19 Increment::Increment( int c, int i )
20     : increment( i ) // initializer for const member
21 { count = c; }
22
23 // Print the data
24 void Increment::print() const
25 {
26     cout << "count = " << count
27         << ", increment = " << increment << endl;
28 }
29
30 int main()
31 {
32     Increment value( 10, 5 );
33
34     cout << "Before incrementing: ";
35     value.print();
36
37     for ( int j = 0; j < 3; j++ ) {
38         value.addIncrement();
39         cout << "After increment " << j << ": ";
40         value.print();
41     }
42
43     return 0;
44 }

```

```

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.

```

1 // Fig. 7.3: fig07_03.cpp
2 // Attempting to initialize a constant of
3 // a built-in data type with an assignment.
4 #include <iostream.h>
5
6 class Increment {
7 public:
8     Increment( int c = 0, int i = 1 );
9     void addIncrement() { count += increment; }
10    void print() const;
11 private:
12    int count;
13    const int increment;
14 };
15
16 // Constructor for class Increment
17 Increment::Increment( int c, int i )
18 {
19     // Constant member 'increment' is not initialized
20     count = c;
21     increment = i; // ERROR: Cannot modify a const object
22 }
23
24 // Print the data
25 void Increment::print() const
26 {
27     cout << "count = " << count
28         << ", increment = " << increment << endl;
29 }
30
31 int main()
32 {
33     Increment value( 10, 5 );
34
35     cout << "Before incrementing: ";
36     value.print();
37
38     for ( int j = 0; j < 3; j++ ) {
39         value.addIncrement();
40         cout << "After increment " << j << ": ";
41         value.print();
42     }
43
44     return 0;
45 }

```

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).

```

1 // Fig. 7.4: date1.h
2 // Declaration of the Date class.
3 // Member functions defined in date1.cpp
4 #ifndef DATE1_H
5 #define DATE1_H
6
7 class Date {
8 public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to test proper day for month and year
18    int checkDay( int );
19 };
20
21 #endif

```

Fig. 7.4 Using member-object initializers (part 1 of 6).

```

22 // Fig. 7.4: date.cpp
23 // Member function definitions for Date class.
24 #include <iostream.h>
25 #include "date1.h"
26
27 // Constructor: Confirm proper value for month;
28 // call utility function checkDay to confirm proper
29 // value for day.
30 Date::Date( int mn, int dy, int yr )
31 {
32     if ( mn > 0 && mn <= 12 ) // validate the month
33         month = mn;
34     else {
35         month = 1;
36         cout << "Month " << mn << " invalid. Set to month 1.\n";
37     }
38
39     year = yr; // should validate yr
40     day = checkDay( dy ); // validate the day
41
42     cout << "Date object constructor for date ";
43     print(); // interesting: a print with no arguments
44     cout << endl;
45 }
46
47 // Print Date object in form month/day/year

```



```

48 void Date::print() const
49     { cout << month << '/' << day << '/' << year; }
50
51 // Destructor: provided to confirm destruction order
52 Date::~Date()
53 {
54     cout << "Date object destructor for date ";
55     print();
56     cout << endl;
57 }
58

```

Fig. 7.4 Using member-object initializers (part 2 of 6).

```

59 // Utility function to confirm proper day value
60 // based on month and year.
61 // Is the year 2000 a leap year?
62 int Date::checkDay( int testDay )
63 {
64     static const int daysPerMonth[ 13 ] =
65         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
66
67     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
68         return testDay;
69
70     if ( month == 2 && // February: Check for leap year
71         testDay == 29 &&
72         ( year % 400 == 0 || // year 2000?
73           ( year % 4 == 0 && year % 100 != 0 ) ) ) // year 2000?
74         return testDay;
75
76     cout << "Day " << testDay << " invalid. Set to day 1.\n";
77
78     return 1; // leave object in consistent state if bad value
79 }

```

Fig. 7.4 Using member-object initializers (part 3 of 6).

```

80 // Fig. 7.4: emply1.h
81 // Declaration of the Employee class.
82 // Member functions defined in emply1.cpp
83 #ifndef EMPLOY1_H
84 #define EMPLOY1_H
85
86 #include "date1.h"
87
88 class Employee {
89 public:
90     Employee( char *, char *, int, int, int, int, int, int );
91     void print() const;
92     ~Employee(); // provided to confirm destruction order
93 private:
94     char firstName[ 25 ];
95     char lastName[ 25 ];
96     const Date birthDate;
97     const Date hireDate;
98 };
99
100 #endif

```

Fig. 7.4 Using member-object initializers (part 4 of 6).

```

101 // Fig. 7.4: emp1y1.cpp
102 // Member function definitions for Employee class.
103 #include <iostream.h>
104 #include <string.h>
105 #include "emp1y1.h"
106 #include "date1.h"
107
108 Employee::Employee( char *fname, char *lname,
109                   int bmonth, int bday, int byear,
110                   int hmonth, int hday, int hyear )
111   : birthDate( bmonth, bday, byear ),
112     hireDate( hmonth, hday, hyear )
113 {
114   // copy fname into firstName and be sure that it fits
115   int length = strlen( fname );
116   length = ( length < 25 ? length : 24 );
117   strncpy( firstName, fname, length );
118   firstName[ length ] = '\0';
119
120   // copy lname into lastName and be sure that it fits
121   length = strlen( lname );
122   length = ( length < 25 ? length : 24 );
123   strncpy( lastName, lname, length );
124   lastName[ length ] = '\0';
125
126   cout << "Employee object constructor: "
127         << firstName << " " << lastName << endl;
128 }
129
130 void Employee::print() const
131 {
132   cout << lastName << ", " << firstName << "\nHired: ";
133   hireDate.print();
134   cout << " Birth date: ";
135   birthDate.print();
136   cout << endl;
137 }
138
139 // Destructor: provided to confirm destruction order
140 Employee::~Employee()
141 {
142   cout << "Employee object destructor: "
143         << lastName << ", " << firstName << endl;
144 }

```

Fig. 7.4 Using member-object initializers (part 5 of 6).

```

145 // Fig. 7.4: fig07_04.cpp
146 // Demonstrating composition: an object with member objects.
147 #include <iostream.h>
148 #include "emp1y1.h"
149
150 int main()
151 {
152   Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
153
154   cout << '\n';
155   e.print();
156
157   cout << "\nTest Date constructor with invalid values:\n";
158   Date d( 14, 35, 1994 ); // invalid Date values
159   cout << endl;
160   return 0;
161 }

```

```

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).

```

1 // Fig. 7.5: fig07_05.cpp
2 // Friends can access private members of a class.
3 #include <iostream.h>
4
5 // Modified Count class
6 class Count {
7     friend void setX( Count &, int ); // friend declaration
8 public:
9     Count() { x = 0; } // constructor
10    void print() const { cout << x << endl; } // output
11 private:
12    int x; // data member
13 };
14
15 // Can modify private data of Count because
16 // setX is declared as a friend function of Count
17 void setX( Count &c, int val )
18 {
19     c.x = val; // legal: setX is a friend of Count
20 }
21
22 int main()
23 {
24     Count counter;
25
26     cout << "counter.x after instantiation: ";
27     counter.print();
28     cout << "counter.x after call to setX friend function: ";
29     setX( counter, 8 ); // set x with a friend
30     counter.print();
31     return 0;
32 }

```

```

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.

```

1 // Fig. 7.6: fig07_06.cpp
2 // Non-friend/non-member functions cannot access
3 // private data of a class.
4 #include <iostream.h>
5
6 // Modified Count class
7 class Count {
8 public:
9     Count() { x = 0; } // constructor
10    void print() const { cout << x << endl; } // output
11 private:
12    int x; // data member
13 };
14
15 // Function tries to modify private data of Count,
16 // but cannot because it is not a friend of Count.
17 void cannotSetX( Count &c, int val )
18 {
19     c.x = val; // ERROR: 'Count::x' is not accessible
20 }
21
22 int main()
23 {
24     Count counter;
25
26     cannotSetX( counter, 3 ); // cannotSetX is not a friend
27     return 0;
28 }

```

```

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.

```

1 // Fig. 7.7: fig07_07.cpp
2 // Using the this pointer to refer to object members.
3 #include <iostream.h>
4
5 class Test {
6 public:
7     Test( int = 0 ); // default constructor
8     void print() const;
9 private:
10    int x;
11 };
12
13 Test::Test( int a ) { x = a; } // constructor
14

```

Fig. 7.7 Using the **this** pointer (part 1 of 2).

```

15 void Test::print() const    // ( ) around *this required
16 {
17     cout << "          x = " << x
18         << "\n  this->x = " << this->x
19         << "\n(*this).x = " << ( *this ).x << endl;
20 }
21
22 int main()
23 {
24     Test testObject( 12 );
25
26     testObject.print();
27
28     return 0;
29 }

```

```

          x = 12
    this->x = 12
    (*this).x = 12

```

Fig. 7.7 Using the **this** pointer (part 2 of 2).

```

1  // Fig. 7.8: time6.h
2  // Cascading member function calls.
3
4  // Declaration of class Time.
5  // Member functions defined in time6.cpp
6  #ifndef TIME6_H
7  #define TIME6_H
8
9  class Time {
10 public:
11     Time( int = 0, int = 0, int = 0 ); // default constructor
12
13     // set functions
14     Time &setTime( int, int, int ); // set hour, minute, second
15     Time &setHour( int ); // set hour
16     Time &setMinute( int ); // set minute
17     Time &setSecond( int ); // set second
18
19     // get functions (normally declared const)
20     int getHour() const; // return hour
21     int getMinute() const; // return minute
22     int getSecond() const; // return second
23
24     // print functions (normally declared const)
25     void printMilitary() const; // print military time
26     void printStandard() const; // print standard time
27 private:
28     int hour; // 0 - 23
29     int minute; // 0 - 59
30     int second; // 0 - 59
31 };
32
33 #endif

```

Fig. 7.8 Cascading member function calls (part 1 of 4).

```
34 // Fig. 7.8: time.cpp
35 // Member function definitions for Time class.
36 #include "time6.h"
37 #include <iostream.h>
38
39 // Constructor function to initialize private data.
40 // Calls member function setTime to set variables.
41 // Default values are 0 (see class definition).
42 Time::Time( int hr, int min, int sec )
43     { setTime( hr, min, sec ); }
44
45 // Set the values of hour, minute, and second.
46 Time &Time::setTime( int h, int m, int s )
47 {
48     setHour( h );
49     setMinute( m );
50     setSecond( s );
51     return *this; // enables cascading
52 }
53
54 // Set the hour value
55 Time &Time::setHour( int h )
56 {
57     hour = ( h >= 0 && h < 24 ) ? h : 0;
58
59     return *this; // enables cascading
60 }
61
62 // Set the minute value
63 Time &Time::setMinute( int m )
64 {
65     minute = ( m >= 0 && m < 60 ) ? m : 0;
66
67     return *this; // enables cascading
68 }
69
70 // Set the second value
71 Time &Time::setSecond( int s )
72 {
73     second = ( s >= 0 && s < 60 ) ? s : 0;
74
75     return *this; // enables cascading
76 }
77
78 // Get the hour value
79 int Time::getHour() const { return hour; }
80
81 // Get the minute value
82 int Time::getMinute() const { return minute; }
83
```

Fig. 7.8 Cascading member function calls (part 2 of 4).

```

84 // Get the second value
85 int Time::getSecond() const { return second; }
86
87 // Display military format time: HH:MM
88 void Time::printMilitary() const
89 {
90     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
91           << ( minute < 10 ? "0" : "" ) << minute;
92 }
93
94 // Display standard format time: HH:MM:SS AM (or PM)
95 void Time::printStandard() const
96 {
97     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
98           << ":" << ( minute < 10 ? "0" : "" ) << minute
99           << ":" << ( second < 10 ? "0" : "" ) << second
100          << ( hour < 12 ? " AM" : " PM" );
101 }

```

Fig. 7.8 Cascading member function calls (part 3 of 4).

```

102 // Fig. 7.8: fig07_08.cpp
103 // Cascading member function calls together
104 // with the this pointer
105 #include <iostream.h>
106 #include "time6.h"
107
108 int main()
109 {
110     Time t;
111
112     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
113     cout << "Military time: ";
114     t.printMilitary();
115     cout << "\nStandard time: ";
116     t.printStandard();
117
118     cout << "\n\nNew standard time: ";
119     t.setTime( 20, 20, 20 ).printStandard();
120     cout << endl;
121
122     return 0;
123 }

```

```

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).

```

1 // Fig. 7.9: employ1.h
2 // An employee class
3 #ifndef EMPLOY1_H
4 #define EMPLOY1_H
5
6 class Employee {
7 public:
8     Employee( const char*, const char* ); // constructor
9     ~Employee(); // destructor
10    const char *getFirstName() const; // return first name
11    const char *getLastName() const; // return last name
12
13    // static member function
14    static int getCount(); // return # objects instantiated
15
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data member
21    static int count; // number of objects instantiated
22 };
23
24 #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).

```

25 // Fig. 7.9: employ1.cpp
26 // Member function definitions for class Employee
27 #include <iostream.h>
28 #include <string.h>
29 #include <assert.h>
30 #include "employ1.h"
31
32 // Initialize the static data member
33 int Employee::count = 0;
34
35 // Define the static member function that
36 // returns the number of employee objects instantiated.
37 int Employee::getCount() { return count; }
38
39 // Constructor dynamically allocates space for the
40 // first and last name and uses strcpy to copy
41 // the first and last names into the object
42 Employee::Employee( const char *first, const char *last )
43 {
44     firstName = new char[ strlen( first ) + 1 ];
45     assert( firstName != 0 ); // ensure memory allocated
46     strcpy( firstName, first );
47
48     lastName = new char[ strlen( last ) + 1 ];
49     assert( lastName != 0 ); // ensure memory allocated
50     strcpy( lastName, last );
51
52     ++count; // increment static count of employees
53     cout << "Employee constructor for " << firstName
54          << ' ' << lastName << " called." << endl;
55 }
56
57 // Destructor deallocates dynamically allocated memory
58 Employee::~Employee()
59 {
60     cout << "~Employee() called for " << firstName
61          << ' ' << lastName << endl;

```



```

62     delete [] firstName; // recapture memory
63     delete [] lastName; // recapture memory
64     --count; // decrement static count of employees
65 }
66
67 // Return first name of employee
68 const char *Employee::getFirstName() const
69 {
70     // const before return type prevents client modifying
71     // private data. Client should copy returned string before
72     // destructor deletes storage to prevent undefined pointer.
73     return firstName;
74 }

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 2 of 5).

```

75
76 // Return last name of employee
77 const char *Employee::getLastName() const
78 {
79     // const before return type prevents client modifying
80     // private data. Client should copy returned string before
81     // destructor deletes storage to prevent undefined pointer.
82     return lastName;
83 }

```

Fig. 7.9 Using a **static** data member to maintain a count of the number of objects of a class (part 3 of 5).

```

84 // Fig. 7.9: fig07_09.cpp
85 // Driver to test the Employee class
86 #include <iostream.h>
87 #include "employ1.h"
88
89 int main()
90 {
91     cout << "Number of employees before instantiation is "
92         << Employee::getCount() << endl; // use class name
93
94     Employee *e1Ptr = new Employee( "Susan", "Baker" );
95     Employee *e2Ptr = new Employee( "Robert", "Jones" );
96
97     cout << "Number of employees after instantiation is "
98         << e1Ptr->getCount();
99
100    cout << "\n\nEmployee 1: "
101        << e1Ptr->getFirstName()
102        << " " << e1Ptr->getLastName()
103        << "\nEmployee 2: "
104        << e2Ptr->getFirstName()
105        << " " << e2Ptr->getLastName() << "\n\n";
106
107    delete e1Ptr; // recapture memory
108    e1Ptr = 0;
109    delete e2Ptr; // recapture memory
110    e2Ptr = 0;
111
112    cout << "Number of employees after deletion is "
113        << Employee::getCount() << endl;
114
115    return 0;
116 }

```

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).

```

1 // Fig. 7.10: implementation.h
2 // Header file for class Implementation
3
4 class Implementation {
5     public:
6         Implementation( int v ) { value = v; }
7         void setValue( int v ) { value = v; }
8         int getValue() const { return value; }
9
10    private:
11        int value;
12 };

```

Fig. 7.10 Implementing a proxy class (part 1 of 4).

```

13 // Fig. 7.10: interface.h
14 // Header file for interface.cpp
15 class Implementation; // forward class declaration
16
17 class Interface {
18     public:
19         Interface( int );
20         void setValue( int ); // same public interface as
21         int getValue() const; // class Implementation
22     private:
23         Implementation *ptr; // requires previous
24                               // forward declaration
25 };

```

Fig. 7.10 Implementing a proxy class (part 2 of 4).

```
26 // Fig. 7.10: interface.cpp
27 // Definition of class Interface
28 #include "interface.h"
29 #include "implementation.h"
30
31 Interface::Interface( int v )
32     : ptr ( new Implementation( v ) ) { }
33
34 // call Implementation's setValue function
35 void Interface::setValue( int v ) { ptr->setValue( v ); }
36
37 // call Implementation's getValue function
38 int Interface::getValue() const { return ptr->getValue(); }
```

Fig. 7.10 Implementing a proxy class (part 3 of 4).

```
39 // Fig. 7.10: fig07_10.cpp
40 // Hiding a class's private data with a proxy class.
41 #include <iostream.h>
42 #include "interface.h"
43
44 int main()
45 {
46     Interface i( 5 );
47
48     cout << "Interface contains: " << i.getValue()
49         << " before setValue" << endl;
50     i.setValue( 10 );
51     cout << "Interface contains: " << i.getValue()
52         << " after setValue" << endl;
53     return 0;
54 }
```

```
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)**

- Fig. 8.1** Operators that can be overloaded.
- Fig. 8.2** Operators that cannot be overloaded.
- Fig. 8.3** User-defined stream-insertion and stream-extraction operators.
- Fig. 8.4** Demonstrating an **Array** class with overloaded operators.
- Fig. 8.5** Definition of a basic **String** class.
- Fig. 8.6** Output from driver for class **Date**.
- Fig. 8.7** Demonstrating class **Complex**.
- Fig. 8.8** A user-defined huge integer class.

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.

```

1 // Fig. 8.3: fig08_03.cpp
2 // Overloading the stream-insertion and
3 // stream-extraction operators.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 class PhoneNumber {
8     friend ostream &operator<<( ostream&, const PhoneNumber & );
9     friend istream &operator>>( istream&, PhoneNumber & );
10
11 private:
12     char areaCode[ 4 ]; // 3-digit area code and null
13     char exchange[ 4 ]; // 3-digit exchange and null
14     char line[ 5 ];     // 4-digit line and null
15 };

```

Fig. 8.3 User-defined stream-insertion and stream-extraction operators (part 1 of 2).

```

16
17 // Overloaded stream-insertion operator (cannot be
18 // a member function if we would like to invoke it with
19 // cout << somePhoneNumber;).
20 ostream &operator<<( ostream &output, const PhoneNumber &num )
21 {
22     output << "(" << num.areaCode << " ) "
23         << num.exchange << "-" << num.line;
24     return output; // enables cout << a << b << c;
25 }
26
27 istream &operator>>( istream &input, PhoneNumber &num )
28 {
29     input.ignore(); // skip (
30     input >> setw( 4 ) >> num.areaCode; // input area code
31     input.ignore( 2 ); // skip ) and space
32     input >> setw( 4 ) >> num.exchange; // input exchange
33     input.ignore(); // skip dash (-)
34     input >> setw( 5 ) >> num.line; // input line
35     return input; // enables cin >> a >> b >> c;
36 }
37
38 int main()
39 {
40     PhoneNumber phone; // create object phone
41
42     cout << "Enter phone number in the form (123) 456-7890:\n";
43
44     // cin >> phone invokes operator>> function by
45     // issuing the call operator>>( cin, phone ).
46     cin >> phone;
47
48     // cout << phone invokes operator<< function by
49     // issuing the call operator<<( cout, phone ).
50     cout << "The phone number entered was: " << phone << endl;
51     return 0;
52 }

```

```

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).

```

1 // Fig. 8.4: array1.h
2 // Simple class Array (for integers)
3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5
6 #include <iostream.h>
7
8 class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10    friend istream &operator>>( istream &, Array & );
11 public:
12    Array( int = 10 ); // default constructor
13    Array( const Array & ); // copy constructor
14    ~Array(); // destructor
15    int getSize() const; // return size
16    const Array &operator=( const Array & ); // assign arrays
17    bool operator==( const Array & ) const; // compare equal
18
19    // Determine if two arrays are not equal and
20    // return true, otherwise return false (uses operator==).
21    bool operator!=( const Array &right ) const
22    { return ! ( *this == right ); }
23
24    int &operator[]( int ); // subscript operator
25    const int &operator[]( int ) const; // subscript operator
26    static int getArrayCount(); // Return count of
27    // arrays instantiated.
28 private:
29    int size; // size of the array
30    int *ptr; // pointer to first element of array
31    static int arrayCount; // # of Arrays instantiated
32 };
33
34 #endif

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 1 of 8).

```

35 // Fig 8.4: array1.cpp
36 // Member function definitions for class Array
37 #include <iostream.h>
38 #include <iomanip.h>
39 #include <stdlib.h>
40 #include <assert.h>
41 #include "array1.h"
42
43 // Initialize static data member at file scope
44 int Array::arrayCount = 0; // no objects yet
45
46 // Default constructor for class Array (default size 10)
47 Array::Array( int arraySize )
48 {
49     size = ( arraySize > 0 ? arraySize : 10 );
50     ptr = new int[ size ]; // create space for array
51     assert( ptr != 0 ); // terminate if memory not allocated
52     ++arrayCount; // count one more object
53
54     for ( int i = 0; i < size; i++ )
55         ptr[ i ] = 0; // initialize array
56 }
57
58 // Copy constructor for class Array
59 // must receive a reference to prevent infinite recursion
60 Array::Array( const Array &init ) : size( init.size )
61 {

```

```

62     ptr = new int[ size ]; // create space for array
63     assert( ptr != 0 );    // terminate if memory not allocated
64     ++arrayCount;         // count one more object
65
66     for ( int i = 0; i < size; i++ )
67         ptr[ i ] = init.ptr[ i ]; // copy init into object
68 }
69
70 // Destructor for class Array
71 Array::~Array()
72 {
73     delete [] ptr;         // reclaim space for array
74     --arrayCount;         // one fewer objects
75 }
76
77 // Get the size of the array
78 int Array::getSize() const { return size; }
79
80 // Overloaded assignment operator
81 // const return avoids: ( a1 = a2 ) = a3
82 const Array &Array::operator=( const Array &right )
83 {
84     if ( &right != this ) { // check for self-assignment
85

```

Fig. 8.4 Demonstrating an `Array` class with overloaded operators (part 2 of 8).

```

86         // for arrays of different sizes, deallocate original
87         // left side array, then allocate new left side array.
88         if ( size != right.size ) {
89             delete [] ptr; // reclaim space
90             size = right.size; // resize this object
91             ptr = new int[ size ]; // create space for array copy
92             assert( ptr != 0 ); // terminate if not allocated
93         }
94
95         for ( int i = 0; i < size; i++ )
96             ptr[ i ] = right.ptr[ i ]; // copy array into object
97     }
98
99     return *this; // enables x = y = z;
100 }
101
102 // Determine if two arrays are equal and
103 // return true, otherwise return false.
104 bool Array::operator==( const Array &right ) const
105 {
106     if ( size != right.size )
107         return false; // arrays of different sizes
108
109     for ( int i = 0; i < size; i++ )
110         if ( ptr[ i ] != right.ptr[ i ] )
111             return false; // arrays are not equal
112
113     return true; // arrays are equal
114 }
115
116 // Overloaded subscript operator for non-const Arrays
117 // reference return creates an lvalue
118 int &Array::operator[]( int subscript )
119 {
120     // check for subscript out of range error
121     assert( 0 <= subscript && subscript < size );
122

```



```

123     return ptr[ subscript ]; // reference return
124 }
125
126 // Overloaded subscript operator for const Arrays
127 // const reference return creates an rvalue
128 const int &Array::operator[]( int subscript ) const
129 {
130     // check for subscript out of range error
131     assert( 0 <= subscript && subscript < size );
132
133     return ptr[ subscript ]; // const reference return
134 }
135

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 3 of 8).

```

136 // Return the number of Array objects instantiated
137 // static functions cannot be const
138 int Array::getArrayCount() { return arrayCount; }
139
140 // Overloaded input operator for class Array;
141 // inputs values for entire array.
142 istream &operator>>( istream &input, Array &a )
143 {
144     for ( int i = 0; i < a.size; i++ )
145         input >> a.ptr[ i ];
146
147     return input; // enables cin >> x >> y;
148 }
149
150 // Overloaded output operator for class Array
151 ostream &operator<<( ostream &output, const Array &a )
152 {
153     int i;
154
155     for ( i = 0; i < a.size; i++ ) {
156         output << setw( 12 ) << a.ptr[ i ];
157
158         if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
159             output << endl;
160     }
161
162     if ( i % 4 != 0 )
163         output << endl;
164
165     return output; // enables cout << x << y;
166 }

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 4 of 8).

```

167 // Fig. 8.4: fig08_04.cpp
168 // Driver for simple class Array
169 #include <iostream.h>
170 #include "array1.h"
171
172 int main()
173 {
174     // no objects yet
175     cout << "# of arrays instantiated = "
176         << Array::getArrayCount() << '\n';
177
178     // create two arrays and print Array count
179     Array integers1( 7 ), integers2;
180     cout << "# of arrays instantiated = "
181         << Array::getArrayCount() << "\n\n";
182

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 5 of 8).

```

183     // print integers1 size and contents
184     cout << "Size of array integers1 is "
185         << integers1.getSize()
186         << "\nArray after initialization:\n"
187         << integers1 << '\n';
188
189     // print integers2 size and contents
190     cout << "Size of array integers2 is "
191         << integers2.getSize()
192         << "\nArray after initialization:\n"
193         << integers2 << '\n';
194
195     // input and print integers1 and integers2
196     cout << "Input 17 integers:\n";
197     cin >> integers1 >> integers2;
198     cout << "After input, the arrays contain:\n"
199         << "integers1:\n" << integers1
200         << "integers2:\n" << integers2 << '\n';
201
202     // use overloaded inequality (!=) operator
203     cout << "Evaluating: integers1 != integers2\n";
204     if ( integers1 != integers2 )
205         cout << "They are not equal\n";
206
207     // create array integers3 using integers1 as an
208     // initializer; print size and contents
209     Array integers3( integers1 );
210
211     cout << "\nSize of array integers3 is "
212         << integers3.getSize()
213         << "\nArray after initialization:\n"
214         << integers3 << '\n';
215
216     // use overloaded assignment (=) operator
217     cout << "Assigning integers2 to integers1:\n";
218     integers1 = integers2;
219     cout << "integers1:\n" << integers1
220         << "integers2:\n" << integers2 << '\n';
221
222     // use overloaded equality (==) operator
223     cout << "Evaluating: integers1 == integers2\n";
224     if ( integers1 == integers2 )
225         cout << "They are equal\n\n";
226
227     // use overloaded subscript operator to create rvalue

```

```

228     cout << "integers1[5] is " << integers1[5] << '\n';
229
230     // use overloaded subscript operator to create lvalue
231     cout << "Assigning 1000 to integers1[5]\n";
232     integers1[5] = 1000;
233     cout << "integers1:\n" << integers1 << '\n';

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 6 of 8).

```

234
235     // attempt to use out of range subscript
236     cout << "Attempt to assign 1000 to integers1[15]" << endl;
237     integers1[15] = 1000; // ERROR: out of range
238
239     return 0;
240 }

```

```

# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
      0      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).

```

1 // Fig. 8.5: string1.h
2 // Definition of a String class
3 #ifndef STRING1_H
4 #define STRING1_H
5
6 #include <iostream.h>
7
8 class String {
9     friend ostream &operator<<( ostream &, const String & );
10    friend istream &operator>>( istream &, String & );
11
12 public:
13     String( const char * = "" ); // conversion/default ctor
14     String( const String & );    // copy constructor
15     ~String();                  // destructor
16     const String &operator=( const String & ); // assignment
17     const String &operator+=( const String & ); // concatenation
18     bool operator!() const;     // is String empty?
19     bool operator==( const String & ) const; // test s1 == s2
20     bool operator<( const String & ) const; // test s1 < s2
21
22     // test s1 != s2
23     bool operator!=( const String & right ) const
24         { return !( *this == right ); }
25
26     // test s1 > s2
27     bool operator>( const String &right ) const
28         { return right < *this; }
29

```

Fig. 8.5 Definition of a basic **String** class (part 1 of 9).

```

30     // test s1 <= s2
31     bool operator<=( const String &right ) const
32         { return !( right < *this ); }
33
34     // test s1 >= s2
35     bool operator>=( const String &right ) const
36         { return !( *this < right ); }
37
38     char &operator[]( int );           // subscript operator
39     const char &operator[]( int ) const; // subscript operator
40     String &operator()( int, int ); // return a substring
41     int getLength() const;           // return string length
42
43 private:
44     int length;                       // string length
45     char *sPtr;                       // pointer to start of string
46
47     void setString( const char * ); // utility function
48 };
49
50 #endif

```

Fig. 8.5 Definition of a basic **String** class (part 2 of 9).

```

51 // Fig. 8.5: string1.cpp
52 // Member function definitions for class String
53 #include <iostream.h>
54 #include <iomanip.h>
55 #include <string.h>
56 #include <assert.h>
57 #include "string1.h"
58
59 // Conversion constructor: Convert char * to String
60 String::String( const char *s ) : length( strlen( s ) )
61 {
62     cout << "Conversion constructor: " << s << '\n';
63     setString( s ); // call utility function
64 }
65
66 // Copy constructor
67 String::String( const String &copy ) : length( copy.length )
68 {
69     cout << "Copy constructor: " << copy.sPtr << '\n';
70     setString( copy.sPtr ); // call utility function
71 }
72
73 // Destructor
74 String::~String()
75 {
76     cout << "Destructor: " << sPtr << '\n';

```

Fig. 8.5 Definition of a basic **String** class (part 3 of 9).

```

77     delete [] sPtr;          // reclaim string
78 }
79
80 // Overloaded = operator; avoids self assignment
81 const String &String::operator=( const String &right )
82 {
83     cout << "operator= called\n";
84
85     if ( &right != this ) {          // avoid self assignment
86         delete [] sPtr;              // prevents memory leak
87         length = right.length;      // new String length
88         setString( right.sPtr );    // call utility function
89     }
90     else
91         cout << "Attempted assignment of a String to itself\n";
92
93     return *this;    // enables cascaded assignments
94 }
95
96 // Concatenate right operand to this object and
97 // store in this object.
98 const String &String::operator+=( const String &right )
99 {
100     char *tempPtr = sPtr;          // hold to be able to delete
101     length += right.length;        // new String length
102     sPtr = new char[ length + 1 ]; // create space
103     assert( sPtr != 0 );           // terminate if memory not allocated
104     strcpy( sPtr, tempPtr );       // left part of new String
105     strcat( sPtr, right.sPtr );    // right part of new String
106     delete [] tempPtr;             // reclaim old space
107     return *this;                  // enables cascaded calls
108 }
109
110 // Is this String empty?
111 bool String::operator!() const { return length == 0; }
112
113 // Is this String equal to right String?
114 bool String::operator==( const String &right ) const
115     { return strcmp( sPtr, right.sPtr ) == 0; }
116
117 // Is this String less than right String?
118 bool String::operator<( const String &right ) const
119     { return strcmp( sPtr, right.sPtr ) < 0; }
120
121 // Return a reference to a character in a String as an lvalue.
122 char &String::operator[]( int subscript )
123 {
124     // First test for subscript out of range
125     assert( subscript >= 0 && subscript < length );
126

```

Fig. 8.5 Definition of a basic **String** class (part 4 of 9).

```

127     return sPtr[ subscript ]; // creates lvalue
128 }
129
130 // Return a reference to a character in a String as an rvalue.
131 const char &String::operator[]( int subscript ) const
132 {
133     // First test for subscript out of range
134     assert( subscript >= 0 && subscript < length );
135
136     return sPtr[ subscript ]; // creates rvalue
137 }
138
139 // Return a substring beginning at index and
140 // of length subLength as a reference to a String object.
141 String &String::operator()( int index, int subLength )
142 {
143     // ensure index is in range and substring length >= 0
144     assert( index >= 0 && index < length && subLength >= 0 );
145
146     String *subPtr = new String; // empty String
147     assert( subPtr != 0 ); // ensure new String allocated
148
149     // determine length of substring
150     if ( ( subLength == 0 ) || ( index + subLength > length ) )
151         subPtr->length = length - index + 1;
152     else
153         subPtr->length = subLength + 1;
154
155     // allocate memory for substring
156     delete subPtr->sPtr; // delete character array from object
157     subPtr->sPtr = new char[ subPtr->length ];
158     assert( subPtr->sPtr != 0 ); // ensure space allocated
159
160     // copy substring into new String
161     strncpy( subPtr->sPtr, &sPtr[ index ], subPtr->length );
162     subPtr->sPtr[ subPtr->length ] = '\0'; // terminate String
163
164     return *subPtr; // return new String
165 }
166
167 // Return string length
168 int String::getLength() const { return length; }
169
170 // Utility function to be called by constructors and
171 // assignment operator.
172 void String::setString( const char *string2 )
173 {
174     sPtr = new char[ length + 1 ]; // allocate storage
175     assert( sPtr != 0 ); // terminate if memory not allocated
176     strcpy( sPtr, string2 ); // copy literal to object
177 }

```

Fig. 8.5 Definition of a basic **String** class (part 5 of 9).

```

178
179 // Overloaded output operator
180 ostream &operator<<( ostream &output, const String &s )
181 {
182     output << s.sPtr;
183     return output;    // enables cascading
184 }
185
186 // Overloaded input operator
187 istream &operator>>( istream &input, String &s )
188 {
189     char temp[ 100 ];    // buffer to store input
190
191     input >> setw( 100 ) >> temp;
192     s = temp;           // use String class assignment operator
193     return input;      // enables cascading
194 }

```

Fig. 8.5 Member function definitions for class **String** (part 6 of 9).

```

195 // Fig. 8.5: fig08_05.cpp
196 // Driver for class String
197 #include <iostream.h>
198 #include "string1.h"
199
200 int main()
201 {
202     String s1( "happy" ), s2( " birthday" ), s3;
203
204     // test overloaded equality and relational operators
205     cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
206         << "\"; s3 is \"" << s3 << "\"\n";
207     << "\nThe results of comparing s2 and s1:"
208     << "\ns2 == s1 yields "
209     << ( s2 == s1 ? "true" : "false" )
210     << "\ns2 != s1 yields "
211     << ( s2 != s1 ? "true" : "false" )
212     << "\ns2 > s1 yields "
213     << ( s2 > s1 ? "true" : "false" )
214     << "\ns2 < s1 yields "
215     << ( s2 < s1 ? "true" : "false" )
216     << "\ns2 >= s1 yields "
217     << ( s2 >= s1 ? "true" : "false" )
218     << "\ns2 <= s1 yields "
219     << ( s2 <= s1 ? "true" : "false" );
220
221     // test overloaded String empty (!) operator
222     cout << "\n\nTesting !s3:\n";
223     if ( !s3 ) {
224         cout << "s3 is empty; assigning s1 to s3;\n";

```

Fig. 8.5 Definition of a basic **String** class (part 7 of 9).


```

225     s3 = s1;           // test overloaded assignment
226     cout << "s3 is \"" << s3 << "\"";
227 }
228
229 // test overloaded String concatenation operator
230 cout << "\n\ns1 += s2 yields s1 = ";
231 s1 += s2;           // test overloaded concatenation
232 cout << s1;
233
234 // test conversion constructor
235 cout << "\n\ns1 += \" to you\" yields\n";
236 s1 += " to you";    // test conversion constructor
237 cout << "s1 = " << s1 << "\n\n";
238
239 // test overloaded function call operator () for substring
240 cout << "The substring of s1 starting at\n"
241     << "location 0 for 14 characters, s1(0, 14), is:\n"
242     << s1( 0, 14 ) << "\n\n";
243
244 // test substring "to-end-of-String" option
245 cout << "The substring of s1 starting at\n"
246     << "location 15, s1(15, 0), is: "
247     << s1( 15, 0 ) << "\n\n"; // 0 is "to end of string"
248
249 // test copy constructor
250 String *s4Ptr = new String(s1);
251 cout << "*s4Ptr = " << *s4Ptr << "\n\n";
252
253 // test assignment (=) operator with self-assignment
254 cout << "assigning *s4Ptr to *s4Ptr\n";
255 *s4Ptr = *s4Ptr;    // test overloaded assignment
256 cout << "*s4Ptr = " << *s4Ptr << '\n';
257
258 // test destructor
259 delete s4Ptr;
260
261 // test using subscript operator to create lvalue
262 s1[ 0 ] = 'H';
263 s1[ 6 ] = 'B';
264 cout << "\n\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
265     << s1 << "\n\n";
266
267 // test subscript out of range
268 cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
269 s1[ 30 ] = 'd';    // ERROR: subscript out of range
270
271 return 0;
272 }

```

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).

```

1 // Fig. 8.6: date1.h
2 // Definition of class Date
3 #ifndef DATE1_H
4 #define DATE1_H
5 #include <iostream.h>
6
7 class Date {
8     friend ostream &operator<<( ostream &, const Date & );
9
10 public:
11     Date( int m = 1, int d = 1, int y = 1900 ); // constructor
12     void setDate( int, int, int ); // set the date
13     Date &operator++(); // preincrement operator
14     Date operator++( int ); // postincrement operator
15     const Date &operator+=( int ); // add days, modify object
16     bool leapYear( int ); // is this a leap year?
17     bool endOfMonth( int ); // is this end of month?
18
19 private:
20     int month;
21     int day;
22     int year;
23
24     static const int days[]; // array of days per month
25     void helpIncrement(); // utility function
26 };
27
28 #endif

```

Fig. 8.6 Class **Date** with overloaded increment operators (part 1 of 7).

```

29 // Fig. 8.6: date1.cpp
30 // Member function definitions for Date class
31 #include <iostream.h>
32 #include "date1.h"
33
34 // Initialize static member at file scope;
35 // one class-wide copy.
36 const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
37                             31, 31, 30, 31, 30, 31 };
38
39 // Date constructor
40 Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
41
42 // Set the date
43 void Date::setDate( int mm, int dd, int yy )
44 {
45     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
46     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
47

```

Fig. 8.6 Class **Date** with overloaded increment operators (part 2 of 7).

```

48     // test for a leap year
49     if ( month == 2 && leapYear( year ) )
50         day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
51     else
52         day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
53 }
54
55 // Preincrement operator overloaded as a member function.
56 Date &Date::operator++()
57 {
58     helpIncrement();
59     return *this; // reference return to create an lvalue
60 }
61
62 // Postincrement operator overloaded as a member function.
63 // Note that the dummy integer parameter does not have a
64 // parameter name.
65 Date Date::operator++( int )
66 {
67     Date temp = *this;
68     helpIncrement();
69
70     // return non-incremented, saved, temporary object
71     return temp; // value return; not a reference return
72 }
73
74 // Add a specific number of days to a date
75 const Date &Date::operator+=( int additionalDays )
76 {
77     for ( int i = 0; i < additionalDays; i++ )
78         helpIncrement();
79
80     return *this; // enables cascading
81 }
82
83 // If the year is a leap year, return true;
84 // otherwise, return false
85 bool Date::leapYear( int y )
86 {
87     if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
88         return true; // a leap year
89     else
90         return false; // not a leap year
91 }
92

```

Fig. 8.6 Class `Date` with overloaded increment operators (part 3 of 7).

```

93 // Determine if the day is the end of the month
94 bool Date::endOfMonth( int d )
95 {
96     if ( month == 2 && leapYear( year ) )
97         return d == 29; // last day of Feb. in leap year
98     else
99         return d == days[ month ];
100 }
101
102 // Function to help increment the date
103 void Date::helpIncrement()
104 {
105     if ( endOfMonth( day ) && month == 12 ) { // end year
106         day = 1;
107         month = 1;
108         ++year;

```

```

109     }
110     else if ( endOfMonth( day ) ) {           // end month
111         day = 1;
112         ++month;
113     }
114     else // not end of month or year; increment day
115         ++day;
116 }
117
118 // Overloaded output operator
119 ostream &operator<<( ostream &output, const Date &d )
120 {
121     static char *monthName[ 13 ] = { "", "January",
122         "February", "March", "April", "May", "June",
123         "July", "August", "September", "October",
124         "November", "December" };
125
126     output << monthName[ d.month ] << ' '
127         << d.day << ", " << d.year;
128
129     return output; // enables cascading
130 }

```

Fig. 8.6 Class **Date** with overloaded increment operators (part 4 of 7).

```

131 // Fig. 8.6: fig08_06.cpp
132 // Driver for class Date
133 #include <iostream.h>
134 #include "date1.h"
135
136 int main()
137 {
138     Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );

```

Fig. 8.6 Class **Date** with overloaded increment operators (part 5 of 7).

```

139     cout << "d1 is " << d1
140         << "\nd2 is " << d2
141         << "\nd3 is " << d3 << "\n\n";
142
143     cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
144
145     d3.setDate( 2, 28, 1992 );
146     cout << " d3 is " << d3;
147     cout << "\n++d3 is " << ++d3 << "\n\n";
148
149     Date d4( 3, 18, 1969 );
150
151     cout << "Testing the preincrement operator:\n"
152         << " d4 is " << d4 << '\n';
153     cout << "++d4 is " << ++d4 << '\n';
154     cout << " d4 is " << d4 << "\n\n";
155
156     cout << "Testing the postincrement operator:\n"
157         << " d4 is " << d4 << '\n';
158     cout << "d4++ is " << d4++ << '\n';
159     cout << " d4 is " << d4 << endl;
160
161     return 0;
162 }

```

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).

```

1 // Fig. 8.7: complex1.h
2 // Definition of class Complex
3 #ifndef COMPLEX1_H
4 #define COMPLEX1_H
5
6 class Complex {
7 public:
8     Complex( double = 0.0, double = 0.0 ); // constructor
9     Complex operator+( const Complex & ) const; // addition
10    Complex operator-( const Complex & ) const; // subtraction
11    const Complex &operator=( const Complex & ); // assignment
12    void print() const; // output
13 private:
14    double real; // real part
15    double imaginary; // imaginary part
16 };
17
18 #endif

```

Fig. 8.7 Demonstrating class `Complex` (part 1 of 5).

```

19 // Fig. 8.7: complex1.cpp
20 // Member function definitions for class Complex
21 #include <iostream.h>
22 #include "complex1.h"
23
24 // Constructor
25 Complex::Complex( double r, double i )
26     : real( r ), imaginary( i ) { }
27
28 // Overloaded addition operator
29 Complex Complex::operator+( const Complex &operand2 ) const
30 {
31     return Complex( real + operand2.real,
32                    imaginary + operand2.imaginary );
33 }
34

```

```

35 // Overloaded subtraction operator
36 Complex Complex::operator-( const Complex &operand2 ) const
37 {
38     return Complex( real - operand2.real,
39                   imaginary - operand2.imaginary );
40 }

```

Fig. 8.7 Demonstrating class **Complex** (part 2 of 5).

```

41
42 // Overloaded = operator
43 const Complex& Complex::operator=( const Complex &right )
44 {
45     real = right.real;
46     imaginary = right.imaginary;
47     return *this; // enables cascading
48 }
49
50 // Display a Complex object in the form: (a, b)
51 void Complex::print() const
52 { cout << '(' << real << ", " << imaginary << ')'; }

```

Fig. 8.7 Demonstrating class **Complex** (part 3 of 5).

```

53 // Fig. 8.7: fig08_07.cpp
54 // Driver for class Complex
55 #include <iostream.h>
56 #include "complex1.h"
57
58 int main()
59 {
60     Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );
61
62     cout << "x: ";
63     x.print();
64     cout << "\ny: ";
65     y.print();
66     cout << "\nz: ";
67     z.print();
68
69     x = y + z;
70     cout << "\n\nx = y + z:\n";
71     x.print();
72     cout << " = ";
73     y.print();
74     cout << " + ";
75     z.print();
76
77     x = y - z;
78     cout << "\n\nx = y - z:\n";
79     x.print();
80     cout << " = ";
81     y.print();
82     cout << " - ";
83     z.print();
84     cout << endl;
85
86     return 0;
87 }

```

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).

```

1 // Fig. 8.8: hugeint1.h
2 // Definition of the HugeInt class
3 #ifndef HUGEINT1_H
4 #define HUGEINT1_H
5
6 #include <iostream.h>
7
8 class HugeInt {
9     friend ostream &operator<<( ostream &, HugeInt & );
10 public:
11     HugeInt( long = 0 ); // conversion/default constructor
12     HugeInt( const char * ); // conversion constructor
13     HugeInt operator+( HugeInt & ); // add another HugeInt
14     HugeInt operator+( int ); // add an int
15     HugeInt operator+( const char * ); // add an int in a char *
16 private:
17     short integer[30];
18 };
19
20 #endif

```

Fig. 8.8 A user-defined huge integer class (part 1 of 5).

```

21 // Fig. 8.8: hugeint1.cpp
22 // Member and friend function definitions for class HugeInt
23 #include <string.h>
24 #include "hugeint1.h"
25
26 // Conversion constructor
27 HugeInt::HugeInt( long val )
28 {
29     int i;
30
31     for ( i = 0; i <= 29; i++ )
32         integer[ i ] = 0; // initialize array to zero
33
34     for ( i = 29; val != 0 && i >= 0; i-- ) {
35         integer[ i ] = val % 10;
36         val /= 10;
37     }
38 }
39
40 HugeInt::HugeInt( const char *string )
41 {
42     int i, j;
43

```



```

44     for ( i = 0; i <= 29; i++ )
45         integer[ i ] = 0;
46
47     for ( i = 30 - strlen( string ), j = 0; i <= 29; i++, j++ )
48         integer[ i ] = string[ j ] - '0';
49 }
50
51 // Addition
52 HugeInt HugeInt::operator+( HugeInt &op2 )
53 {
54     HugeInt temp;
55     int carry = 0;
56
57     for ( int i = 29; i >= 0; i-- ) {
58         temp.integer[ i ] = integer[ i ] +
59             op2.integer[ i ] + carry;
60
61         if ( temp.integer[ i ] > 9 ) {
62             temp.integer[ i ] %= 10;
63             carry = 1;
64         }
65         else
66             carry = 0;
67     }
68
69     return temp;
70 }

```

Fig. 8.8 A user-defined huge integer class (part 2 of 5).

```

71
72 // Addition
73 HugeInt HugeInt::operator+( int op2 )
74     { return *this + HugeInt( op2 ); }
75
76 // Addition
77 HugeInt HugeInt::operator+( const char *op2 )
78     { return *this + HugeInt( op2 ); }
79
80 ostream& operator<<( ostream &output, HugeInt &num )
81 {
82     int i;
83
84     for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
85         ; // skip leading zeros
86
87     if ( i == 30 )
88         output << 0;
89     else
90         for ( ; i <= 29; i++ )
91             output << num.integer[ i ];
92
93     return output;
94 }

```

Fig. 8.8 A user-defined huge integer class (part 3 of 5).

Illustrations List [\(Main Page\)](#)

- 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** Casting base-class pointers to derived-class pointers.
- Fig. 9.5** Overriding a base-class member function in a derived class.
- Fig. 9.6** Summary of base-class member accessibility in a derived class.
- Fig. 9.7** Order in which base-class and derived-class constructors and destructors are called.
- Fig. 9.8** Demonstrating class **Point**.
- Fig. 9.9** Demonstrating class **Circle**.
- Fig. 9.10** Demonstrating class **Cylinder**.
- Fig. 9.11** Demonstrating multiple inheritance.

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.

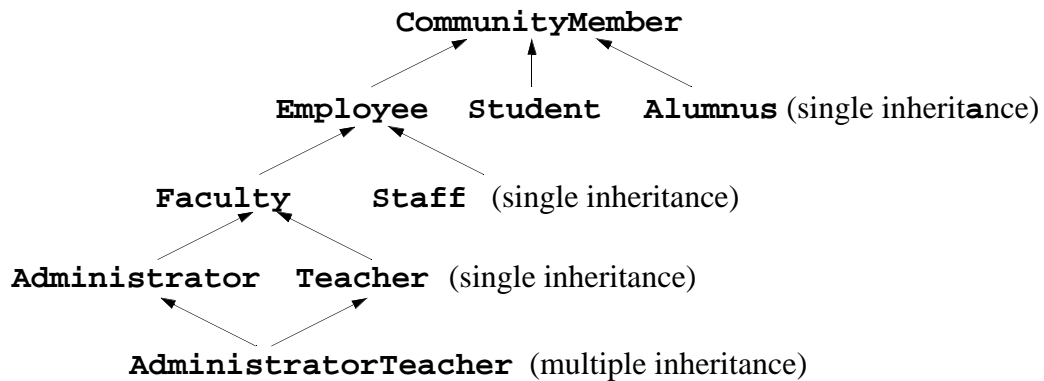


Fig. 9.2 An inheritance hierarchy for university community members.

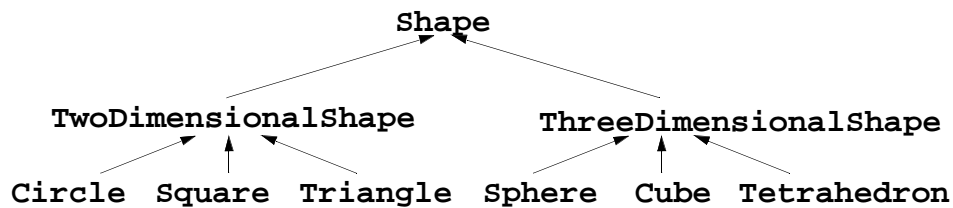


Fig. 9.3 A portion of a Shape class hierarchy.

```

1 // Fig. 9.4: point.h
2 // Definition of class Point
3 #ifndef POINT_H
4 #define POINT_H
5
6 class Point {
7     friend ostream &operator<<( ostream &, const Point & );
8 public:
9     Point( int = 0, int = 0 );        // default constructor
10    void setPoint( int, int );        // set coordinates
11    int getX() const { return x; }    // get x coordinate
12    int getY() const { return y; }    // get y coordinate
13 protected:                          // accessible by derived classes
14    int x, y;                          // x and y coordinates of the Point
15 };
16
17 #endif

```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 1 of 6).

```

18 // Fig. 9.4: point.cpp
19 // Member functions for class Point
20 #include <iostream.h>
21 #include "point.h"
22
23 // Constructor for class Point
24 Point::Point( int a, int b ) { setPoint( a, b ); }
25
26 // Set x and y coordinates of Point
27 void Point::setPoint( int a, int b )
28 {
29     x = a;
30     y = b;
31 }
32
33 // Output Point (with overloaded stream insertion operator)
34 ostream &operator<<( ostream &output, const Point &p )
35 {
36     output << '[' << p.x << ", " << p.y << ']';
37
38     return output;    // enables cascaded calls
39 }

```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 2 of 6).

```

40 // Fig. 9.4: circle.h
41 // Definition of class Circle
42 #ifndef CIRCLE_H
43 #define CIRCLE_H
44
45 #include <iostream.h>
46 #include <iomanip.h>
47 #include "point.h"
48
49 class Circle : public Point { // Circle inherits from Point
50     friend ostream &operator<<( ostream &, const Circle & );
51 public:
52     // default constructor
53     Circle( double r = 0.0, int x = 0, int y = 0 );
54
55     void setRadius( double ); // set radius
56     double getRadius() const; // return radius
57     double area() const;      // calculate area

```

```

58 protected:
59     double radius;
60 };
61
62 #endif

```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 3 of 6).

```

63 // Fig. 9.4: circle.cpp
64 // Member function definitions for class Circle
65 #include "circle.h"
66
67 // Constructor for Circle calls constructor for Point
68 // with a member initializer then initializes radius.
69 Circle::Circle( double r, int a, int b )
70     : Point( a, b )          // call base-class constructor
71 { setRadius( r ); }
72
73 // Set radius of Circle
74 void Circle::setRadius( double r )
75     { radius = ( r >= 0 ? r : 0 ); }
76
77 // Get radius of Circle
78 double Circle::getRadius() const { return radius; }
79
80 // Calculate area of Circle
81 double Circle::area() const
82     { return 3.14159 * radius * radius; }
83
84 // Output a Circle in the form:
85 // Center = [x, y]; Radius = #.##
86 ostream &operator<<( ostream &output, const Circle &c )
87 {
88     output << "Center = " << static_cast< Point >( c )
89         << "; Radius = "
90         << setiosflags( ios::fixed | ios::showpoint )
91         << setprecision( 2 ) << c.radius;
92
93     return output;    // enables cascaded calls
94 }

```

Fig. 9.4 Casting base-class pointers to derived-class pointers (part 4 of 6).

```

95 // Fig. 9.4: fig09_04.cpp
96 // Casting base-class pointers to derived-class pointers
97 #include <iostream.h>
98 #include <iomanip.h>
99 #include "point.h"
100 #include "circle.h"
101
102 int main()
103 {
104     Point *pointPtr = 0, p( 30, 50 );
105     Circle *circlePtr = 0, c( 2.7, 120, 89 );
106
107     cout << "Point p: " << p << "\nCircle c: " << c << '\n';
108
109     // Treat a Circle as a Point (see only the base class part)
110     pointPtr = &c;    // assign address of Circle to pointPtr
111     cout << "\nCircle c (via *pointPtr): "
112         << *pointPtr << '\n';
113
114     // Treat a Circle as a Circle (with some casting)

```

```

115     pointPtr = &c;    // assign address of Circle to pointPtr
116
117     // cast base-class pointer to derived-class pointer
118     circlePtr = static_cast< Circle * >( pointPtr );
119     cout << "\nCircle c (via *circlePtr):\n" << *circlePtr
120           << "\nArea of c (via circlePtr): "
121           << circlePtr->area() << '\n';
122
123     // DANGEROUS: Treat a Point as a Circle
124     pointPtr = &p;    // assign address of Point to pointPtr
125
126     // cast base-class pointer to derived-class pointer
127     circlePtr = static_cast< Circle * >( pointPtr );
128     cout << "\nPoint p (via *circlePtr):\n" << *circlePtr
129           << "\nArea of object circlePtr points to: "
130           << circlePtr->area() << endl;
131     return 0;
132 }

```

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).

```

1 // Fig. 9.5: employ.h
2 // Definition of class Employee
3 #ifndef EMPLOY_H
4 #define EMPLOY_H
5
6 class Employee {
7 public:
8     Employee( const char *, const char * ); // constructor
9     void print() const; // output first and last name
10    ~Employee(); // destructor
11 private:
12    char *firstName; // dynamically allocated string
13    char *lastName; // dynamically allocated string
14 };
15
16 #endif

```

Fig. 9.5 Overriding a base-class member function in a derived class (part 1 of 5).

```

17 // Fig. 9.5: employ.cpp
18 // Member function definitions for class Employee
19 #include <string.h>
20 #include <iostream.h>
21 #include <assert.h>
22 #include "employ.h"
23
24 // Constructor dynamically allocates space for the
25 // first and last name and uses strcpy to copy
26 // the first and last names into the object.
27 Employee::Employee( const char *first, const char *last )
28 {
29     firstName = new char[ strlen( first ) + 1 ];
30     assert( firstName != 0 ); // terminate if not allocated
31     strcpy( firstName, first );
32
33     lastName = new char[ strlen( last ) + 1 ];
34     assert( lastName != 0 ); // terminate if not allocated
35     strcpy( lastName, last );
36 }
37
38 // Output employee name
39 void Employee::print() const
40     { cout << firstName << ' ' << lastName; }
41
42 // Destructor deallocates dynamically allocated memory
43 Employee::~Employee()
44 {
45     delete [] firstName; // reclaim dynamic memory
46     delete [] lastName; // reclaim dynamic memory
47 }

```

Fig. 9.5 Overriding a base-class member function in a derived class (part 2 of 5).

```

48 // Fig. 9.5: hourly.h
49 // Definition of class HourlyWorker
50 #ifndef HOURLY_H
51 #define HOURLY_H
52
53 #include "employ.h"
54
55 class HourlyWorker : public Employee {
56 public:
57     HourlyWorker( const char*, const char*, double, double );
58     double getPay() const; // calculate and return salary
59     void print() const; // overridden base-class print
60 private:
61     double wage; // wage per hour
62     double hours; // hours worked for week
63 };
64
65 #endif

```

Fig. 9.5 Overriding a base-class member function in a derived class (part 3 of 5).


```

66 // Fig. 9.5: hourly.cpp
67 // Member function definitions for class HourlyWorker
68 #include <iostream.h>
69 #include <iomanip.h>
70 #include "hourly.h"
71
72 // Constructor for class HourlyWorker
73 HourlyWorker::HourlyWorker( const char *first,
74                             const char *last,
75                             double initHours, double initWage )
76     : Employee( first, last ) // call base-class constructor
77 {
78     hours = initHours; // should validate
79     wage = initWage;   // should validate
80 }
81
82 // Get the HourlyWorker's pay
83 double HourlyWorker::getPay() const { return wage * hours; }
84
85 // Print the HourlyWorker's name and pay
86 void HourlyWorker::print() const
87 {
88     cout << "HourlyWorker::print() is executing\n\n";
89     Employee::print(); // call base-class print function
90
91     cout << " is an hourly worker with pay of $"
92          << setiosflags( ios::fixed | ios::showpoint )
93          << setprecision( 2 ) << getPay() << endl;
94 }

```

Fig. 9.5 Overriding a base-class member function in a derived class (part 4 of 5).

```

95 // Fig. 9.5: fig.09_05.cpp
96 // Overriding a base-class member function in a
97 // derived class.
98 #include <iostream.h>
99 #include "hourly.h"
100
101 int main()
102 {
103     HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
104     h.print();
105     return 0;
106 }

```

```

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.	protected in derived class.	private in derived class.
	Can be accessed directly by any non- static member functions, friend functions and non-member functions.	Can be accessed directly by all non- static member functions and friend functions.	Can be accessed directly by all non- static member functions and friend functions.
protected	protected in derived class.	protected in derived class.	private in derived class.
	Can be accessed directly by all non- static member functions and friend functions.	Can be accessed directly by all non- static member functions and friend functions.	Can be accessed directly by all non- static member functions and friend functions.
private	Hidden in derived class.	Hidden in derived 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.	Can be accessed by non- static member functions and friend functions through public or protected member functions of the base 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.

```

1 // Fig. 9.7: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 class Point {
7 public:
8     Point( int = 0, int = 0 ); // default constructor
9     ~Point(); // destructor
10 protected: // accessible by derived classes
11     int x, y; // x and y coordinates of Point
12 };
13
14 #endif

```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 1 of 5).

```

15 // Fig. 9.7: point2.cpp
16 // Member function definitions for class Point
17 #include <iostream.h>
18 #include "point2.h"
19
20 // Constructor for class Point
21 Point::Point( int a, int b )
22 {
23     x = a;
24     y = b;
25
26     cout << "Point constructor: "
27         << '[' << x << ", " << y << ']' << endl;
28 }
29
30 // Destructor for class Point
31 Point::~~Point()
32 {
33     cout << "Point destructor: "
34         << '[' << x << ", " << y << ']' << endl;
35 }

```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 2 of 5).

```

36 // Fig. 9.7: circle2.h
37 // Definition of class Circle
38 #ifndef CIRCLE2_H
39 #define CIRCLE2_H
40
41 #include "point2.h"
42
43 class Circle : public Point {
44 public:
45     // default constructor
46     Circle( double r = 0.0, int x = 0, int y = 0 );
47
48     ~Circle();
49 private:
50     double radius;
51 };
52
53 #endif

```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 3 of 5).

```

54 // Fig. 9.7: circle2.cpp
55 // Member function definitions for class Circle
56 #include "circle2.h"
57
58 // Constructor for Circle calls constructor for Point
59 Circle::Circle( double r, int a, int b )
60     : Point( a, b ) // call base-class constructor
61 {
62     radius = r; // should validate
63     cout << "Circle constructor: radius is "
64         << radius << " [" << x << ", " << y << "]" << endl;
65 }
66
67 // Destructor for class Circle
68 Circle::~Circle()
69 {
70     cout << "Circle destructor: radius is "
71         << radius << " [" << x << ", " << y << "]" << endl;
72 }

```

Fig. 9.7 Order in which base-class and derived-class constructors and destructors are called (part 4 of 5).

```

73 // Fig. 9.7: fig09_07.cpp
74 // Demonstrate when base-class and derived-class
75 // constructors and destructors are called.
76 #include <iostream.h>
77 #include "point2.h"
78 #include "circle2.h"
79
80 int main()
81 {
82     // Show constructor and destructor calls for Point
83     {
84         Point p( 11, 22 );
85     }
86
87     cout << endl;
88     Circle circle1( 4.5, 72, 29 );
89     cout << endl;
90     Circle circle2( 10, 5, 5 );
91     cout << endl;
92     return 0;
93 }

```

```

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).

```

1 // Fig. 9.8: point2.h
2 // Definition of class Point
3 #ifndef POINT2_H
4 #define POINT2_H
5
6 class Point {
7     friend ostream &operator<<( ostream &, const Point & );
8 public:
9     Point( int = 0, int = 0 ); // default constructor
10    void setPoint( int, int ); // set coordinates
11    int getX() const { return x; } // get x coordinate
12    int getY() const { return y; } // get y coordinate
13 protected: // accessible to derived classes
14    int x, y; // coordinates of the point
15 };
16
17 #endif

```

Fig. 9.8 Demonstrating class **Point** (part 1 of 3).

```

18 // Fig. 9.8: point2.cpp
19 // Member functions for class Point
20 #include <iostream.h>
21 #include "point2.h"
22
23 // Constructor for class Point
24 Point::Point( int a, int b ) { setPoint( a, b ); }
25
26 // Set the x and y coordinates
27 void Point::setPoint( int a, int b )
28 {
29     x = a;
30     y = b;
31 }
32
33 // Output the Point
34 ostream &operator<<( ostream &output, const Point &p )
35 {
36     output << '[' << p.x << ", " << p.y << ']' ;
37
38     return output; // enables cascading
39 }

```

Fig. 9.8 Demonstrating class **Point** (part 2 of 3).

```

40 // Fig. 9.8: fig09_08.cpp
41 // Driver for class Point
42 #include <iostream.h>
43 #include "point2.h"
44
45 int main()
46 {
47     Point p( 72, 115 ); // instantiate Point object p
48
49     // protected data of Point inaccessible to main
50     cout << "X coordinate is " << p.getX()
51          << "\nY coordinate is " << p.getY();
52
53     p.setPoint( 10, 10 );
54     cout << "\n\nThe new location of p is " << p << endl;
55
56     return 0;
57 }

```

```
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).

```
1 // Fig. 9.9: circle2.h
2 // Definition of class Circle
3 #ifndef CIRCLE2_H
4 #define CIRCLE2_H
5
6 #include "point2.h"
7
8 class Circle : public Point {
9     friend ostream &operator<<( ostream &, const Circle & );
10 public:
11     // default constructor
12     Circle( double r = 0.0, int x = 0, int y = 0 );
13     void setRadius( double ); // set radius
14     double getRadius() const; // return radius
15     double area() const; // calculate area
16 protected: // accessible to derived classes
17     double radius; // radius of the Circle
18 };
19
20 #endif
```

Fig. 9.9 Demonstrating class **Circle** (part 1 of 5).

```
21 // Fig. 9.9: circle2.cpp
22 // Member function definitions for class Circle
23 #include <iostream.h>
24 #include <iomanip.h>
25 #include "circle2.h"
26
27 // Constructor for Circle calls constructor for Point
28 // with a member initializer and initializes radius
29 Circle::Circle( double r, int a, int b )
30     : Point( a, b ) // call base-class constructor
31     { setRadius( r ); }
32
33 // Set radius
34 void Circle::setRadius( double r )
35     { radius = ( r >= 0 ? r : 0 ); }
36
37 // Get radius
38 double Circle::getRadius() const { return radius; }
```

Fig. 9.9 Demonstrating class **Circle** (part 2 of 5).

```

39
40 // Calculate area of Circle
41 double Circle::area() const
42     { return 3.14159 * radius * radius; }
43
44 // Output a circle in the form:
45 // Center = [x, y]; Radius = #.##
46 ostream &operator<<( ostream &output, const Circle &c )
47 {
48     output << "Center = " << static_cast< Point > ( c )
49         << "; Radius = "
50         << setiosflags( ios::fixed | ios::showpoint )
51         << setprecision( 2 ) << c.radius;
52
53     return output; // enables cascaded calls
54 }

```

Fig. 9.9 Demonstrating class **Circle** (part 3 of 5).

```

55 // Fig. 9.9: fig09_09.cpp
56 // Driver for class Circle
57 #include <iostream.h>
58 #include "point2.h"
59 #include "circle2.h"
60
61 int main()
62 {
63     Circle c( 2.5, 37, 43 );
64
65     cout << "X coordinate is " << c.getX()
66         << "\nY coordinate is " << c.getY()
67         << "\nRadius is " << c.getRadius();
68
69     c.setRadius( 4.25 );
70     c.setPoint( 2, 2 );
71     cout << "\n\nThe new location and radius of c are\n"
72         << c << "\nArea " << c.area() << '\n';
73
74     Point &pRef = c;
75     cout << "\nCircle printed as a Point is: " << pRef << endl;
76
77     return 0;
78 }

```

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).

```

1 // Fig. 9.10: cylindr2.h
2 // Definition of class Cylinder
3 #ifndef CYLINDER2_H
4 #define CYLINDER2_H
5
6 #include "circle2.h"
7
8 class Cylinder : public Circle {
9     friend ostream &operator<<( ostream &, const Cylinder & );
10
11 public:
12     // default constructor
13     Cylinder( double h = 0.0, double r = 0.0,
14             int x = 0, int y = 0 );
15
16     void setHeight( double ); // set height
17     double getHeight() const; // return height
18     double area() const; // calculate and return area
19     double volume() const; // calculate and return volume
20
21 protected:
22     double height; // height of the Cylinder
23 };
24
25 #endif

```

Fig. 9.10 Demonstrating class **Cylinder** (part 1 of 5).

```

26 // Fig. 9.10: cylindr2.cpp
27 // Member and friend function definitions
28 // for class Cylinder.
29 #include <iostream.h>
30 #include <iomanip.h>
31 #include "cylindr2.h"
32

```

Fig. 9.10 Demonstrating class **Cylinder** (part 2 of 5).

```

33 // Cylinder constructor calls Circle constructor
34 Cylinder::Cylinder( double h, double r, int x, int y )
35     : Circle( r, x, y ) // call base-class constructor
36     { setHeight( h ); }
37
38 // Set height of Cylinder
39 void Cylinder::setHeight( double h )
40     { height = ( h >= 0 ? h : 0 ); }
41
42 // Get height of Cylinder
43 double Cylinder::getHeight() const { return height; }
44
45 // Calculate area of Cylinder (i.e., surface area)
46 double Cylinder::area() const
47 {
48     return 2 * Circle::area() +
49         2 * 3.14159 * radius * height;
50 }
51
52 // Calculate volume of Cylinder
53 double Cylinder::volume() const
54     { return Circle::area() * height; }
55
56 // Output Cylinder dimensions
57 ostream &operator<<( ostream &output, const Cylinder &c )

```



```

58 {
59     output << static_cast< Circle >( c )
60         << "; Height = " << c.height;
61
62     return output;    // enables cascaded calls
63 }

```

Fig. 9.10 Demonstrating class **Cylinder** (part 3 of 5).

```

64 // Fig. 9.10: fig09_10.cpp
65 // Driver for class Cylinder
66 #include <iostream.h>
67 #include <iomanip.h>
68 #include "point2.h"
69 #include "circle2.h"
70 #include "cylindr2.h"
71
72 int main()
73 {
74     // create Cylinder object
75     Cylinder cyl( 5.7, 2.5, 12, 23 );
76

```

Fig. 9.10 Demonstrating class **Cylinder** (part 4 of 5).

```

77     // use get functions to display the Cylinder
78     cout << "X coordinate is " << cyl.getX()
79         << "\nY coordinate is " << cyl.getY()
80         << "\nRadius is " << cyl.getRadius()
81         << "\nHeight is " << cyl.getHeight() << "\n\n";
82
83     // use set functions to change the Cylinder's attributes
84     cyl.setHeight( 10 );
85     cyl.setRadius( 4.25 );
86     cyl.setPoint( 2, 2 );
87     cout << "The new location, radius, and height of cyl are:\n"
88         << cyl << '\n';
89
90     // display the Cylinder as a Point
91     Point &pRef = cyl;    // pRef "thinks" it is a Point
92     cout << "\nCylinder printed as a Point is: "
93         << pRef << "\n\n";
94
95     // display the Cylinder as a Circle
96     Circle &circleRef = cyl; // circleRef thinks it is a Circle
97     cout << "Cylinder printed as a Circle is:\n" << circleRef
98         << "\nArea: " << circleRef.area() << endl;
99
100    return 0;
101 }

```

```

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).

```

1 // Fig. 9.11: base1.h
2 // Definition of class Base1
3 #ifndef BASE1_H
4 #define BASE1_H
5
6 class Base1 {
7 public:
8     Base1( int x ) { value = x; }
9     int getData() const { return value; }
10 protected: // accessible to derived classes
11     int value; // inherited by derived class
12 };
13
14 #endif

```

Fig. 9.11 Demonstrating multiple inheritance (part 1 of 6).

```

15 // Fig. 9.11: base2.h
16 // Definition of class Base2
17 #ifndef BASE2_H
18 #define BASE2_H
19
20 class Base2 {
21 public:
22     Base2( char c ) { letter = c; }
23     char getData() const { return letter; }
24 protected: // accessible to derived classes
25     char letter; // inherited by derived class
26 };
27
28 #endif

```

Fig. 9.11 Demonstrating multiple inheritance (part 2 of 6).

```

29 // Fig. 9.11: derived.h
30 // Definition of class Derived which inherits
31 // multiple base classes (Base1 and Base2).
32 #ifndef DERIVED_H
33 #define DERIVED_H
34
35 #include "base1.h"
36 #include "base2.h"
37
38 // multiple inheritance
39 class Derived : public Base1, public Base2 {
40     friend ostream &operator<<( ostream &, const Derived & );
41
42 public:
43     Derived( int, char, double );
44     double getReal() const;
45
46 private:
47     double real;    // derived class's private data
48 };
49
50 #endif

```

Fig. 9.11 Demonstrating multiple inheritance (part 3 of 6).

```

51 // Fig. 9.11: derived.cpp
52 // Member function definitions for class Derived
53 #include <iostream.h>
54 #include "derived.h"
55
56 // Constructor for Derived calls constructors for
57 // class Base1 and class Base2.
58 // Use member initializers to call base-class constructors
59 Derived::Derived( int i, char c, double f )
60     : Base1( i ), Base2( c ), real ( f ) { }
61
62 // Return the value of real
63 double Derived::getReal() const { return real; }
64
65 // Display all the data members of Derived
66 ostream &operator<<( ostream &output, const Derived &d )
67 {
68     output << "    Integer: " << d.value
69         << "\n Character: " << d.letter
70         << "\nReal number: " << d.real;
71
72     return output;    // enables cascaded calls
73 }

```

Fig. 9.11 Demonstrating multiple inheritance (part 4 of 6).

```

74 // Fig. 9.11: fig09_11.cpp
75 // Driver for multiple inheritance example
76 #include <iostream.h>
77 #include "base1.h"
78 #include "base2.h"
79 #include "derived.h"
80
81 int main()
82 {
83     Base1 b1( 10 ), *base1Ptr = 0; // create Base1 object
84     Base2 b2( 'Z' ), *base2Ptr = 0; // create Base2 object
85     Derived d( 7, 'A', 3.5 );      // create Derived object
86
87     // print data members of base class objects
88     cout << "Object b1 contains integer " << b1.getData()
89         << "\nObject b2 contains character " << b2.getData()
90         << "\nObject d contains:\n" << d << "\n\n";
91
92     // print data members of derived class object
93     // scope resolution operator resolves getData ambiguity
94     cout << "Data members of Derived can be"
95         << " accessed individually:"
96         << "\n Integer: " << d.Base1::getData()
97         << "\n Character: " << d.Base2::getData()
98         << "\nReal number: " << d.getReal() << "\n\n";
99
100    cout << "Derived can be treated as an "
101        << "object of either base class:\n";
102
103    // treat Derived as a Base1 object
104    base1Ptr = &d;
105    cout << "base1Ptr->getData() yields "
106        << base1Ptr->getData() << '\n';
107
108    // treat Derived as a Base2 object
109    base2Ptr = &d;
110    cout << "base2Ptr->getData() yields "
111        << base2Ptr->getData() << endl;
112
113    return 0;
114 }

```

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** Demonstrating polymorphism with the **Employee** class hierarchy.
Fig. 10.2 Definition of abstract base class **Shape**.
Fig. 10.3 Flow of control of a **virtual** function call.

```

1 // Fig. 10.1: employ2.h
2 // Abstract base class Employee
3 #ifndef EMPLOY2_H
4 #define EMPLOY2_H
5
6 #include <iostream.h>
7
8 class Employee {
9 public:
10     Employee( const char *, const char * );
11     ~Employee(); // destructor reclaims memory
12     const char *getFirstName() const;
13     const char *getLastName() const;
14
15     // Pure virtual function makes Employee abstract base class
16     virtual double earnings() const = 0; // pure virtual
17     virtual void print() const; // virtual
18 private:
19     char *firstName;
20     char *lastName;
21 };
22
23 #endif

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 1 of 13).

```

24 // Fig. 10.1: employ2.cpp
25 // Member function definitions for
26 // abstract base class Employee.
27 // Note: No definitions given for pure virtual functions.
28 #include <string.h>
29 #include <assert.h>
30 #include "employ2.h"
31
32 // Constructor dynamically allocates space for the
33 // first and last name and uses strcpy to copy
34 // the first and last names into the object.
35 Employee::Employee( const char *first, const char *last )
36 {
37     firstName = new char[ strlen( first ) + 1 ];
38     assert( firstName != 0 ); // test that new worked
39     strcpy( firstName, first );
40
41     lastName = new char[ strlen( last ) + 1 ];
42     assert( lastName != 0 ); // test that new worked
43     strcpy( lastName, last );
44 }
45
46 // Destructor deallocates dynamically allocated memory
47 Employee::~Employee()
48 {
49     delete [] firstName;
50     delete [] lastName;
51 }
52
53 // Return a pointer to the first name
54 // Const return type prevents caller from modifying private
55 // data. Caller should copy returned string before destructor
56 // deletes dynamic storage to prevent undefined pointer.
57 const char *Employee::getFirstName() const
58 {
59     return firstName; // caller must delete memory
60 }

```

```

61
62 // Return a pointer to the last name
63 // Const return type prevents caller from modifying private
64 // data. Caller should copy returned string before destructor
65 // deletes dynamic storage to prevent undefined pointer.
66 const char *Employee::getLastName() const
67 {
68     return lastName; // caller must delete memory
69 }
70
71 // Print the name of the Employee
72 void Employee::print() const
73     { cout << firstName << ' ' << lastName; }

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 2 of 13).

```

74 // Fig. 10.1: boss1.h
75 // Boss class derived from Employee
76 #ifndef BOSS1_H
77 #define BOSS1_H
78 #include "employ2.h"
79
80 class Boss : public Employee {
81 public:
82     Boss( const char *, const char *, double = 0.0 );
83     void setWeeklySalary( double );
84     virtual double earnings() const;
85     virtual void print() const;
86 private:
87     double weeklySalary;
88 };
89
90 #endif

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 3 of 13).

```

91 // Fig. 10.1: boss1.cpp
92 // Member function definitions for class Boss
93 #include "boss1.h"
94
95 // Constructor function for class Boss
96 Boss::Boss( const char *first, const char *last, double s )
97     : Employee( first, last ) // call base-class constructor
98     { setWeeklySalary( s ); }
99
100 // Set the Boss's salary
101 void Boss::setWeeklySalary( double s )
102     { weeklySalary = s > 0 ? s : 0; }
103
104 // Get the Boss's pay
105 double Boss::earnings() const { return weeklySalary; }
106
107 // Print the Boss's name
108 void Boss::print() const
109 {
110     cout << "\n                Boss: ";
111     Employee::print();
112 }

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 4 of 13).


```

113 // Fig. 10.1: commis1.h
114 // CommissionWorker class derived from Employee
115 #ifndef COMMIS1_H
116 #define COMMIS1_H
117 #include "employ2.h"
118
119 class CommissionWorker : public Employee {
120 public:
121     CommissionWorker( const char *, const char *,
122                     double = 0.0, double = 0.0,
123                     int = 0 );
124     void setSalary( double );
125     void setCommission( double );
126     void setQuantity( int );
127     virtual double earnings() const;
128     virtual void print() const;
129 private:
130     double salary;           // base salary per week
131     double commission;     // amount per item sold
132     int quantity;          // total items sold for week
133 };
134
135 #endif

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 5 of 13).

```

136 // Fig. 10.1: commis1.cpp
137 // Member function definitions for class CommissionWorker
138 #include <iostream.h>
139 #include "commis1.h"
140
141 // Constructor for class CommissionWorker
142 CommissionWorker::CommissionWorker( const char *first,
143                                   const char *last, double s, double c, int q )
144     : Employee( first, last ) // call base-class constructor
145 {
146     setSalary( s );
147     setCommission( c );
148     setQuantity( q );
149 }
150
151 // Set CommissionWorker's weekly base salary
152 void CommissionWorker::setSalary( double s )
153     { salary = s > 0 ? s : 0; }
154
155 // Set CommissionWorker's commission
156 void CommissionWorker::setCommission( double c )
157     { commission = c > 0 ? c : 0; }
158
159 // Set CommissionWorker's quantity sold
160 void CommissionWorker::setQuantity( int q )
161     { quantity = q > 0 ? q : 0; }
162
163 // Determine CommissionWorker's earnings
164 double CommissionWorker::earnings() const
165     { return salary + commission * quantity; }
166
167 // Print the CommissionWorker's name
168 void CommissionWorker::print() const
169 {
170     cout << "\nCommission worker: ";
171     Employee::print();

```

```
172 }
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 6 of 13).

```
173 // Fig. 10.1: piece1.h
174 // PieceWorker class derived from Employee
175 #ifndef PIECE1_H
176 #define PIECE1_H
177 #include "employ2.h"
178
179 class PieceWorker : public Employee {
180 public:
181     PieceWorker( const char *, const char *,
182                double = 0.0, int = 0);
183     void setWage( double );
184     void setQuantity( int );
185     virtual double earnings() const;
186     virtual void print() const;
187 private:
188     double wagePerPiece; // wage for each piece output
189     int quantity;       // output for week
190 };
191
192 #endif
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 7 of 13).

```
193 // Fig. 10.1: piece1.cpp
194 // Member function definitions for class PieceWorker
195 #include <iostream.h>
196 #include "piece1.h"
197
198 // Constructor for class PieceWorker
199 PieceWorker::PieceWorker( const char *first, const char *last,
200                          double w, int q )
201     : Employee( first, last ) // call base-class constructor
202 {
203     setWage( w );
204     setQuantity( q );
205 }
206
207 // Set the wage
208 void PieceWorker::setWage( double w )
209     { wagePerPiece = w > 0 ? w : 0; }
210
211 // Set the number of items output
212 void PieceWorker::setQuantity( int q )
213     { quantity = q > 0 ? q : 0; }
214
215 // Determine the PieceWorker's earnings
216 double PieceWorker::earnings() const
217     { return quantity * wagePerPiece; }
```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 8 of 13).

```
218
219 // Print the PieceWorker's name
220 void PieceWorker::print() const
221 {
222     cout << "\n    Piece worker: ";
```

```

223     Employee::print();
224 }

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 9 of 13).

```

225 // Fig. 10.1: hourly1.h
226 // Definition of class HourlyWorker
227 #ifndef HOURLY1_H
228 #define HOURLY1_H
229 #include "employ2.h"
230
231 class HourlyWorker : public Employee {
232 public:
233     HourlyWorker( const char *, const char *,
234                 double = 0.0, double = 0.0);
235     void setWage( double );
236     void setHours( double );
237     virtual double earnings() const;
238     virtual void print() const;
239 private:
240     double wage;    // wage per hour
241     double hours;  // hours worked for week
242 };
243
244 #endif

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 10 of 13).

```

1 // Fig. 10.1: hourly1.cpp
2 // Member function definitions for class HourlyWorker
3 #include <iostream.h>
4 #include "hourly1.h"
5
6 // Constructor for class HourlyWorker
7 HourlyWorker::HourlyWorker( const char *first,
8                             const char *last,
9                             double w, double h )
10 : Employee( first, last ) // call base-class constructor
11 {
12     setWage( w );
13     setHours( h );
14 }
15
16 // Set the wage
17 void HourlyWorker::setWage( double w )
18     { wage = w > 0 ? w : 0; }
19
20 // Set the hours worked
21 void HourlyWorker::setHours( double h )
22     { hours = h >= 0 && h < 168 ? h : 0; }
23
24 // Get the HourlyWorker's pay
25 double HourlyWorker::earnings() const
26 {
27     if ( hours <= 40 ) // no overtime
28         return wage * hours;
29     else // overtime is paid at wage * 1.5
30         return 40 * wage + ( hours - 40 ) * wage * 1.5;
31 }
32
33 // Print the HourlyWorker's name

```

```

34 void HourlyWorker::print() const
35 {
36     cout << "\n    Hourly worker: ";
37     Employee::print();
38 }

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 11 of 13).

```

39 // Fig. 10.1: fig10_01.cpp
40 // Driver for Employee hierarchy
41 #include <iostream.h>
42 #include <iomanip.h>
43 #include "employ2.h"
44 #include "boss1.h"
45 #include "commis1.h"
46 #include "piece1.h"
47 #include "hourly1.h"
48
49 void virtualViaPointer( const Employee * );
50 void virtualViaReference( const Employee & );
51
52 int main()
53 {
54     // set output formatting
55     cout << setiosflags( ios::fixed | ios::showpoint )
56          << setprecision( 2 );
57
58     Boss b( "John", "Smith", 800.00 );
59     b.print(); // static binding
60     cout << " earned $" << b.earnings(); // static binding
61     virtualViaPointer( &b ); // uses dynamic binding
62     virtualViaReference( b ); // uses dynamic binding
63
64     CommissionWorker c( "Sue", "Jones", 200.0, 3.0, 150 );
65     c.print(); // static binding
66     cout << " earned $" << c.earnings(); // static binding
67     virtualViaPointer( &c ); // uses dynamic binding
68     virtualViaReference( c ); // uses dynamic binding
69
70     PieceWorker p( "Bob", "Lewis", 2.5, 200 );
71     p.print(); // static binding
72     cout << " earned $" << p.earnings(); // static binding
73     virtualViaPointer( &p ); // uses dynamic binding
74     virtualViaReference( p ); // uses dynamic binding
75
76     HourlyWorker h( "Karen", "Price", 13.75, 40 );
77     h.print(); // static binding
78     cout << " earned $" << h.earnings(); // static binding
79     virtualViaPointer( &h ); // uses dynamic binding
80     virtualViaReference( h ); // uses dynamic binding
81     cout << endl;
82     return 0;
83 }
84

```

Fig. 10.1 Demonstrating polymorphism with the **Employee** class hierarchy (part 12 of 13).

```

85 // Make virtual function calls off a base-class pointer
86 // using dynamic binding.
87 void virtualViaPointer( const Employee *baseClassPtr )
88 {
89     baseClassPtr->print();
90     cout << " earned $" << baseClassPtr->earnings();
91 }
92
93 // Make virtual function calls off a base-class reference
94 // using dynamic binding.
95 void virtualViaReference( const Employee &baseClassRef )
96 {
97     baseClassRef.print();
98     cout << " earned $" << baseClassRef.earnings();
99 }

```

```

          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).

```

1 // Fig. 10.2: shape.h
2 // Definition of abstract base class Shape
3 #ifndef SHAPE_H
4 #define SHAPE_H
5 #include <iostream.h>
6
7 class Shape {
8 public:
9     virtual double area() const { return 0.0; }
10    virtual double volume() const { return 0.0; }
11
12    // pure virtual functions overridden in derived classes
13    virtual void printShapeName() const = 0;
14    virtual void print() const = 0;
15 };
16
17 #endif

```

Fig. 10.2 Definition of abstract base class **Shape** (part 1 of 10).

```

18 // Fig. 10.2: point1.h
19 // Definition of class Point
20 #ifndef POINT1_H
21 #define POINT1_H
22 #include "shape.h"
23
24 class Point : public Shape {
25 public:

```

```

26     Point( int = 0, int = 0 ); // default constructor
27     void setPoint( int, int );
28     int getX() const { return x; }
29     int getY() const { return y; }
30     virtual void printShapeName() const { cout << "Point: "; }
31     virtual void print() const;
32 private:
33     int x, y; // x and y coordinates of Point
34 };
35
36 #endif

```

Fig. 10.2 Definition of class **Point** (part 2 of 10).

```

37 // Fig. 10.2: point1.cpp
38 // Member function definitions for class Point
39 #include "point1.h"
40
41 Point::Point( int a, int b ) { setPoint( a, b ); }
42
43 void Point::setPoint( int a, int b )
44 {
45     x = a;
46     y = b;
47 }
48
49 void Point::print() const
50 { cout << '[' << x << ", " << y << ']'<< endl; }

```

Fig. 10.2 Member function definitions for class **Point** (part 3 of 10).

```

51 // Fig. 10.2: circle1.h
52 // Definition of class Circle
53 #ifndef CIRCLE1_H
54 #define CIRCLE1_H
55 #include "point1.h"
56
57 class Circle : public Point {
58 public:
59     // default constructor
60     Circle( double r = 0.0, int x = 0, int y = 0 );
61
62     void setRadius( double );
63     double getRadius() const;
64     virtual double area() const;
65     virtual void printShapeName() const { cout << "Circle: "; }
66     virtual void print() const;
67 private:
68     double radius; // radius of Circle
69 };
70
71 #endif

```

Fig. 10.2 Definition of class **Circle** (part 4 of 10).

```

72 // Fig. 10.2: circle1.cpp
73 // Member function definitions for class Circle
74 #include "circle1.h"
75
76 Circle::Circle( double r, int a, int b )
77     : Point( a, b ) // call base-class constructor
78     { setRadius( r ); }
79

```

```

80 void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }
81
82 double Circle::getRadius() const { return radius; }
83
84 double Circle::area() const
85     { return 3.14159 * radius * radius; }
86
87 void Circle::print() const
88     {
89     Point::print();
90     cout << " ; Radius = " << radius;
91     }

```

Fig. 10.2 Member function definitions for class **Circle** (part 5 of 10).

```

1 // Fig. 10.2: cylindr1.h
2 // Definition of class Cylinder
3 #ifndef CYLINDR1_H
4 #define CYLINDR1_H
5 #include "circle1.h"
6
7 class Cylinder : public Circle {
8 public:
9     // default constructor
10    Cylinder( double h = 0.0, double r = 0.0,
11            int x = 0, int y = 0 );
12
13    void setHeight( double );
14    double getHeight() const;
15    virtual double area() const;
16    virtual double volume() const;
17    virtual void printShapeName() const {cout << "Cylinder: ";}
18    virtual void print() const;
19 private:
20    double height; // height of Cylinder
21 };
22
23 #endif

```

Fig. 10.2 Definition of class **Cylinder** (part 6 of 10).

```

24 // Fig. 10.2: cylindr1.cpp
25 // Member and friend function definitions for class Cylinder
26 #include "cylindr1.h"
27
28 Cylinder::Cylinder( double h, double r, int x, int y )
29     : Circle( r, x, y ) // call base-class constructor
30     { setHeight( h ); }
31
32 void Cylinder::setHeight( double h )
33     { height = h > 0 ? h : 0; }
34
35 double Cylinder::getHeight() const { return height; }
36
37 double Cylinder::area() const
38     {
39     // surface area of Cylinder
40     return 2 * Circle::area() +
41            2 * 3.14159 * getRadius() * height;
42     }
43
44 double Cylinder::volume() const
45     { return Circle::area() * height; }

```

```
46
47 void Cylinder::print() const
48 {
49     Circle::print();
50     cout << "; Height = " << height;
51 }
```

Fig. 10.2 Member function definitions for class **Cylinder** (part 7 of 10).

```
52 // Fig. 10.2: fig10_02.cpp
53 // Driver for shape, point, circle, cylinder hierarchy
54 #include <iostream.h>
55 #include <iomanip.h>
56 #include "shape.h"
57 #include "point1.h"
58 #include "circle1.h"
59 #include "cylindr1.h"
60
61 void virtualViaPointer( const Shape * );
62 void virtualViaReference( const Shape & );
63
64 int main()
65 {
66     cout << setiosflags( ios::fixed | ios::showpoint )
67         << setprecision( 2 );
68 }
```

Fig. 10.2 Driver for point, circle, cylinder hierarchy (part 8 of 10).


```

69     Point point( 7, 11 );                // create a Point
70     Circle circle( 3.5, 22, 8 );        // create a Circle
71     Cylinder cylinder( 10, 3.3, 10, 10 ); // create a Cylinder
72
73     point.printShapeName();             // static binding
74     point.print();                      // static binding
75     cout << '\n';
76
77     circle.printShapeName();            // static binding
78     circle.print();                    // static binding
79     cout << '\n';
80
81     cylinder.printShapeName();          // static binding
82     cylinder.print();                  // static binding
83     cout << "\n\n";
84
85     Shape *arrayOfShapes[ 3 ];          // array of base-class pointers
86
87     // aim arrayOfShapes[0] at derived-class Point object
88     arrayOfShapes[ 0 ] = &point;
89
90     // aim arrayOfShapes[1] at derived-class Circle object
91     arrayOfShapes[ 1 ] = &circle;
92
93     // aim arrayOfShapes[2] at derived-class Cylinder object
94     arrayOfShapes[ 2 ] = &cylinder;
95
96     // Loop through arrayOfShapes and call virtualViaPointer
97     // to print the shape name, attributes, area, and volume
98     // of each object using dynamic binding.
99     cout << "Virtual function calls made off "
100         << "base-class pointers\n";
101
102     for ( int i = 0; i < 3; i++ )
103         virtualViaPointer( arrayOfShapes[ i ] );
104
105     // Loop through arrayOfShapes and call virtualViaReference
106     // to print the shape name, attributes, area, and volume
107     // of each object using dynamic binding.
108     cout << "Virtual function calls made off "
109         << "base-class references\n";
110
111     for ( int j = 0; j < 3; j++ )
112         virtualViaReference( *arrayOfShapes[ j ] );
113
114     return 0;
115 }
116

```

Fig. 10.2 Driver for point, circle, cylinder hierarchy (part 9 of 10).

```

117 // Make virtual function calls off a base-class pointer
118 // using dynamic binding.
119 void virtualViaPointer( const Shape *baseClassPtr )
120 {
121     baseClassPtr->printShapeName();
122     baseClassPtr->print();
123     cout << "\nArea = " << baseClassPtr->area()
124         << "\nVolume = " << baseClassPtr->volume() << "\n\n";
125 }
126
127 // Make virtual function calls off a base-class reference
128 // using dynamic binding.
129 void virtualViaReference( const Shape &baseClassRef )
130 {
131     baseClassRef.printShapeName();
132     baseClassRef.print();
133     cout << "\nArea = " << baseClassRef.area()
134         << "\nVolume = " << baseClassRef.volume() << "\n\n";
135 }

```

```

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).

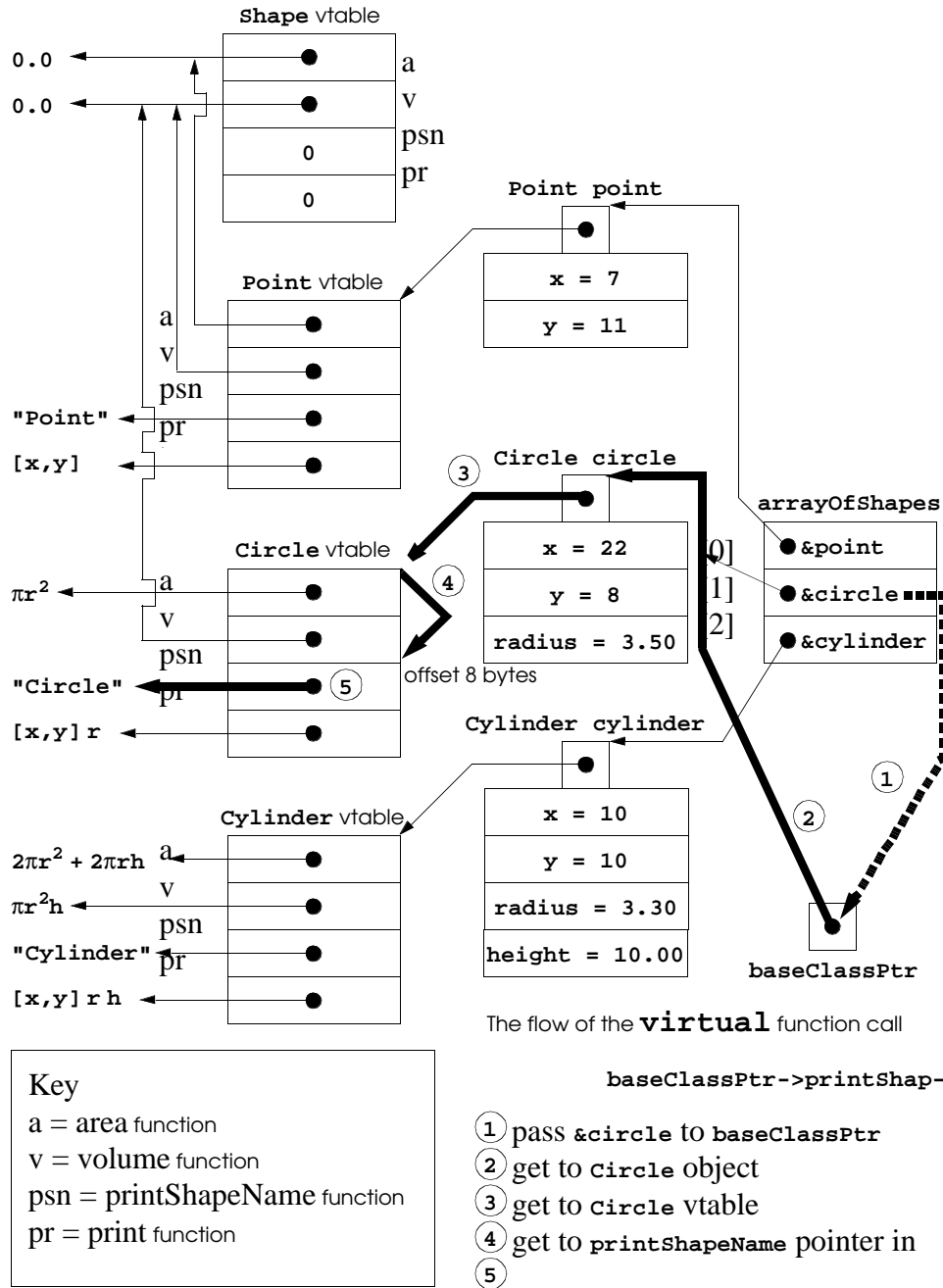


Fig. 10.3 Flow of control of a virtual function call.

Illustrations List [\(Main Page\)](#)

- Fig. 11.1** Portion of stream-I/O class hierarchy.
- Fig. 11.2** Portion of stream-I/O class hierarchy with key file-processing classes.
- Fig. 11.3** Outputting a string using stream insertion.
- Fig. 11.4** Outputting a string using two stream insertions.
- Fig. 11.5** Using the **endl** stream manipulator.
- Fig. 11.6** Outputting expression values.
- Fig. 11.7** Cascading the overloaded **<<** operator.
- Fig. 11.8** Printing the address stored in a **char *** variable.
- Fig. 11.9** Calculating the sum of two integers input from the keyboard with **cin**.
- Fig. 11.10** Avoiding a precedence problem between the stream-insertion operator and the conditional operator.
- Fig. 11.11** Stream-extraction operator returning false on end-of-file.
- Fig. 11.12** Using member functions **get**, **put**, and **eof**.
- Fig. 11.13** Contrasting input of a string using **cin** with stream extraction and input with **cin.get**.
- Fig. 11.14** Character input with member function **getline**.
- Fig. 11.15** Unformatted I/O with the **read**, **gcount** and **write** member functions.
- Fig. 11.16** Using the **hex**, **oct**, **dec** and **setbase** stream manipulators.
- Fig. 11.17** Controlling precision of floating-point values.
- Fig. 11.18** Demonstrating the **width** member function.
- Fig. 11.19** Creating and testing user-defined, nonparameterized stream manipulators.
- Fig. 11.20** Format state flags.
- Fig. 11.21** Controlling the printing of trailing zeros and decimal points with float values.
- Fig. 11.22** Left-justification and right-justification.
- Fig. 11.23** Printing an integer with internal spacing and forcing the plus sign.
- 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.
- Fig. 11.25** Using the **ios::showbase** flag.
- Fig. 11.26** Displaying floating-point values in system default, scientific, and fixed formats.
- Fig. 11.27** Using the **ios::uppercase** flag.
- Fig. 11.28** Demonstrating the **flags** member function.
- Fig. 11.29** Testing error states.

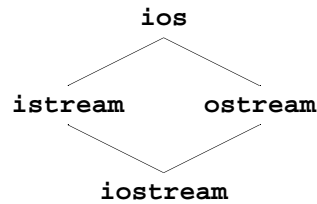


Fig. 11.1 Portion of the stream I/O class hierarchy.

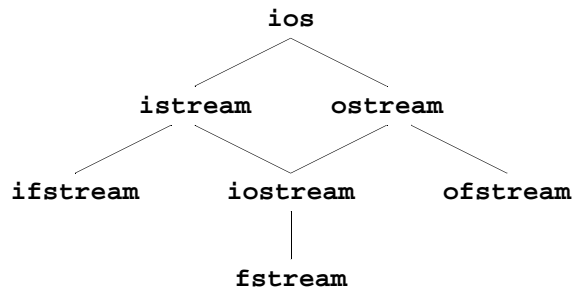


Fig. 11.2 Portion of stream-I/O class hierarchy with key file-processing classes.

```

1 // Fig. 11.3: fig11_03.cpp
2 // Outputting a string using stream insertion.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to C++!\n";
8
9     return 0;
10 }
  
```

Welcome to C++!

Fig. 11.3 Outputting a string using stream insertion.


```
1 // Fig. 11.4: fig11_04.cpp
2 // Outputting a string using two stream insertions.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to ";
8     cout << "C++!\n";
9
10    return 0;
11 }
```



Welcome to C++!

Fig. 11.4 Outputting a string using two stream insertions.

```
1 // Fig. 11.5: fig11_05.cpp
2 // Using the endl stream manipulator.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "Welcome to ";
8     cout << "C++!";
9     cout << endl; // end line stream manipulator
10
11    return 0;
12 }
```



Welcome to C++!

Fig. 11.5 Using the `endl` stream manipulator.

```
1 // Fig. 11.6: fig11_06.cpp
2 // Outputting expression values.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "47 plus 53 is ";
8
9     // parentheses not needed; used for clarity
10    cout << ( 47 + 53 ); // expression
11    cout << endl;
12
13    return 0;
14 }
```



47 plus 53 is 100

Fig. 11.6 Outputting expression values.

```

1 // Fig. 11.7: fig11_07.cpp
2 // Cascading the overloaded << operator.
3 #include <iostream.h>
4
5 int main()
6 {
7     cout << "47 plus 53 is " << ( 47 + 53 ) << endl;
8
9     return 0;
10 }

```

```
47 plus 53 is 100
```

Fig. 11.7 Cascading the overloaded << operator.

```

1 // Fig. 11.8: fig11_08.cpp
2 // Printing the address stored in a char* variable
3 #include <iostream.h>
4
5 int main()
6 {
7     char *string = "test";
8
9     cout << "Value of string is: " << string
10         << "\nValue of static_cast< void *>( string ) is: "
11         << static_cast< void *>( string ) << endl;
12     return 0;
13 }

```

```
Value of string is: test
Value of static_cast< void *>( string ) is: 0x00416D50
```

Fig. 11.8 Printing the address stored in a `char *` variable.

```

1 // Fig. 11.9: fig11_09.cpp
2 // Calculating the sum of two integers input from the keyboard
3 // with the cin object and the stream-extraction operator.
4 #include <iostream.h>
5
6 int main()
7 {
8     int x, y;
9
10    cout << "Enter two integers: ";
11    cin >> x >> y;
12    cout << "Sum of " << x << " and " << y << " is: "
13         << ( x + y ) << endl;
14
15    return 0;
16 }

```

```
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.

```
1 // Fig. 11.10: fig11_10.cpp
2 // Avoiding a precedence problem between the stream-insertion
3 // operator and the conditional operator.
4 // Need parentheses around the conditional expression.
5 #include <iostream.h>
6
7 int main()
8 {
9     int x, y;
10
11     cout << "Enter two integers: ";
12     cin >> x >> y;
13     cout << x << ( x == y ? " is" : " is not" )
14         << " equal to " << y << endl;
15
```

Fig. 11.10 Avoiding a precedence problem between the stream-insertion operator and the conditional operator (part 1 of 2).

```
16     return 0;
17 }
```

```
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).


```

1 // Fig. 11.11: fig11_11.cpp
2 // Stream-extraction operator returning false on end-of-file.
3 #include <iostream.h>
4
5 int main()
6 {
7     int grade, highestGrade = -1;
8
9     cout << "Enter grade (enter end-of-file to end): ";
10    while ( cin >> grade ) {
11        if ( grade > highestGrade )
12            highestGrade = grade;
13
14        cout << "Enter grade (enter end-of-file to end): ";
15    }
16
17    cout << "\n\nHighest grade is: " << highestGrade << endl;
18    return 0;
19 }

```

```

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.

```

1 // Fig. 11.12: fig11_12.cpp
2 // Using member functions get, put, and eof.
3 #include <iostream.h>
4
5 int main()
6 {
7     char c;
8
9     cout << "Before input, cin.eof() is " << cin.eof()
10        << "\nEnter a sentence followed by end-of-file:\n";
11
12    while ( ( c = cin.get() ) != EOF)
13        cout.put( c );
14
15    cout << "\nEOF in this system is: " << c;
16    cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
17    return 0;
18 }

```

```

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`.

```

1 // Fig. 11.13: fig11_13.cpp
2 // Contrasting input of a string with cin and cin.get.
3 #include <iostream.h>
4
5 int main()
6 {
7     const int SIZE = 80;
8     char buffer1[ SIZE ], buffer2[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin >> buffer1;
12    cout << "\nThe string read with cin was:\n"
13         << buffer1 << "\n\n";
14
15    cin.get( buffer2, SIZE );
16    cout << "The string read with cin.get was:\n"
17         << buffer2 << endl;
18
19    return 0;
20 }

```

```

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`.

```

1 // Fig. 11.14: fig11_14.cpp
2 // Character input with member function getline.
3 #include <iostream.h>
4
5 int main()
6 {
7     const SIZE = 80;
8     char buffer[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin.getline( buffer, SIZE );
12
13    cout << "\nThe sentence entered is:\n" << buffer << endl;
14    return 0;
15 }

```

```

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).

```

1 // Fig. 11.15: fig11_15.cpp
2 // Unformatted I/O with read, gcount and write.
3 #include <iostream.h>
4
5 int main()
6 {
7     const int SIZE = 80;
8     char buffer[ SIZE ];
9
10    cout << "Enter a sentence:\n";
11    cin.read( buffer, 20 );
12    cout << "\nThe sentence entered was:\n";
13    cout.write( buffer, cin.gcount() );
14    cout << endl;
15    return 0;
16 }

```

```

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.

```

1 // Fig. 11.16: fig11_16.cpp
2 // Using hex, oct, dec and setbase stream manipulators.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int n;
9
10    cout << "Enter a decimal number: ";
11    cin >> n;
12
13    cout << n << " in hexadecimal is: "
14         << hex << n << '\n'
15         << dec << n << " in octal is: "
16         << oct << n << '\n'
17         << setbase( 10 ) << n << " in decimal is: "
18         << n << endl;
19
20    return 0;
21 }

```

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).

```

1 // Fig. 11.17: fig11_17.cpp
2 // Controlling precision of floating-point values
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <math.h>
6
7 int main()
8 {
9     double root2 = sqrt( 2.0 );
10    int places;
11
12    cout << setiosflags( ios::fixed)
13         << "Square root of 2 with precisions 0-9.\n"
14         << "Precision set by the "
15         << "precision member function:" << endl;
16
17    for ( places = 0; places <= 9; places++ ) {
18        cout.precision( places );
19        cout << root2 << '\n';
20    }
21
22    cout << "\nPrecision set by the "
23         << "setprecision manipulator:\n";
24
25    for ( places = 0; places <= 9; places++ )
26        cout << setprecision( places ) << root2 << '\n';
27
28    return 0;
29 }

```

Fig. 11.17 Controlling precision of floating-point values (part 1 of 2).

```

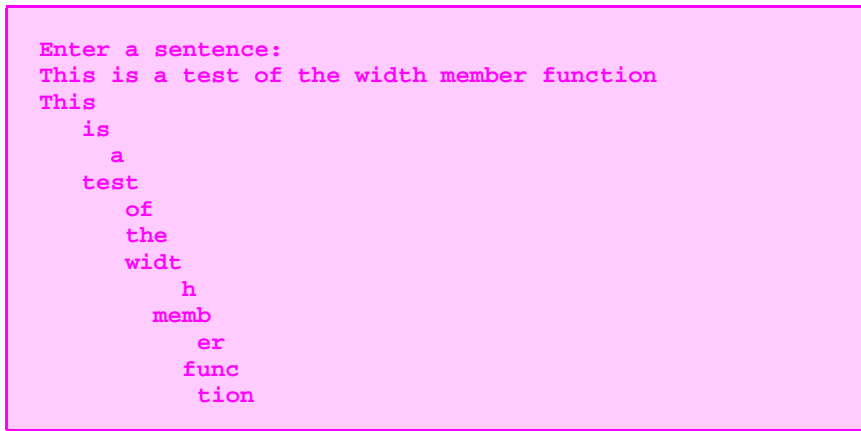
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

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).

```
1 // fig11_18.cpp
2 // Demonstrating the width member function
3 #include <iostream.h>
4
5 int main()
6 {
7     int w = 4;
8     char string[ 10 ];
9
10    cout << "Enter a sentence:\n";
11    cin.width( 5 );
12
13    while ( cin >> string ) {
14        cout.width( w++ );
15        cout << string << endl;
16        cin.width( 5 );
17    }
18
19    return 0;
20 }
```



```
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.

```
1 // Fig. 11.19: fig11_19.cpp
2 // Creating and testing user-defined, nonparameterized
3 // stream manipulators.
4 #include <iostream.h>
5
6 // bell manipulator (using escape sequence \a)
7 ostream& bell( ostream& output ) { return output << '\a'; }
8
9 // ret manipulator (using escape sequence \r)
10 ostream& ret( ostream& output ) { return output << '\r'; }
11
12 // tab manipulator (using escape sequence \t)
13 ostream& tab( ostream& output ) { return output << '\t'; }
14
15 // endLine manipulator (using escape sequence \n
16 // and the flush member function)
17 ostream& endLine( ostream& output )
18 {
19     return output << '\n' << flush;
20 }
21
22 int main()
23 {
24     cout << "Testing the tab manipulator:" << endl
25         << 'a' << tab << 'b' << tab << 'c' << endl
26         << "Testing the ret and bell manipulators:"
27         << endl << ".....";
28     cout << bell;
29     cout << ret << "-----" << endl;
30     return 0;
31 }
```

```
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
<code>ios::skipws</code>	Skip whitespace characters on an input stream.
<code>ios::left</code>	Left justify output in a field. Padding characters appear to the right if necessary.
<code>ios::right</code>	Right justify output in a field. Padding characters appear to the left if necessary.
<code>ios::internal</code>	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).
<code>ios::dec</code>	Specify that integers should be treated as decimal (base 10) values.
<code>ios::oct</code>	Specify that integers should be treated as octal (base 8) values.
<code>ios::hex</code>	Specify that integers should be treated as hexadecimal (base 16) values.
<code>ios::showbase</code>	Specify that the base of a number to be output ahead of the number (a leading <code>0</code> for octals; a leading <code>0x</code> or <code>0X</code> for hexadecimals).
<code>ios::showpoint</code>	Specify that floating-point numbers should be output with a decimal point. This is normally used with <code>ios::fixed</code> to guarantee a certain number of digits to the right of the decimal point.
<code>ios::uppercase</code>	Specify that uppercase <code>x</code> should be used in the <code>0x</code> before a hexadecimal integer and that uppercase <code>E</code> should be used when representing a floating-point value in scientific notation.
<code>ios::showpos</code>	Specify that positive and negative numbers should be preceded by a <code>+</code> or <code>-</code> sign, respectively.
<code>ios::scientific</code>	Specify output of a floating-point value in scientific notation.
<code>ios::fixed</code>	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.

```
1 // Fig. 11.21: fig11_21.cpp
2 // Controlling the printing of trailing zeros and decimal
3 // points for floating-point values.
4 #include <iostream.h>
5 #include <iomanip.h>
6 #include <math.h>
7
8 int main()
9 {
10     cout << "Before setting the ios::showpoint flag\n"
11         << "9.9900 prints as: " << 9.9900
12         << "\n9.9000 prints as: " << 9.9000
13         << "\n9.0000 prints as: " << 9.0000
14         << "\n\nAfter setting the ios::showpoint flag\n";
15     cout.setf( ios::showpoint );
16     cout << "9.9900 prints as: " << 9.9900
17         << "\n9.9000 prints as: " << 9.9000
18         << "\n9.0000 prints as: " << 9.0000 << endl;
19     return 0;
20 }
```

```
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.


```

1 // Fig. 11.22: fig11_22.cpp
2 // Left-justification and right-justification.
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int x = 12345;
9
10    cout << "Default is right justified:\n"
11         << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
12         << "\nUse setf to set ios::left:\n" << setw(10);
13
14    cout.setf( ios::left, ios::adjustfield );
15    cout << x << "\nUse unsetf to restore default:\n";
16    cout.unsetf( ios::left );
17    cout << setw( 10 ) << x
18         << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
19         << "\nUse setiosflags to set ios::left:\n"
20         << setw( 10 ) << setiosflags( ios::left ) << x
21         << "\nUse resetiosflags to restore default:\n"
22         << setw( 10 ) << resetiosflags( ios::left )
23         << x << endl;
24    return 0;
25 }

```

```

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.

```

1 // Fig. 11.23: fig11_23.cpp
2 // Printing an integer with internal spacing and
3 // forcing the plus sign.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 int main()
8 {
9     cout << setiosflags( ios::internal | ios::showpos )
10         << setw( 10 ) << 123 << endl;
11    return 0;
12 }

```

```

+      123

```

Fig. 11.23 Printing an integer with internal spacing and forcing the plus sign.

```

1 // Fig. 11.24: fig11_24.cpp
2 // Using the fill member function and the setfill
3 // manipulator to change the padding character for
4 // fields larger than the values being printed.
5 #include <iostream.h>
6 #include <iomanip.h>
7
8 int main()
9 {
10     int x = 10000;
11
12     cout << x << " printed as int right and left justified\n"
13         << "and as hex with internal justification.\n"
14         << "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).

```

15     cout.setf( ios::showbase );
16     cout << setw( 10 ) << x << '\n';
17     cout.setf( ios::left, ios::adjustfield );
18     cout << setw( 10 ) << x << '\n';
19     cout.setf( ios::internal, ios::adjustfield );
20     cout << setw( 10 ) << hex << x;
21
22     cout << "\n\nUsing various padding characters:\n";
23     cout.setf( ios::right, ios::adjustfield );
24     cout.fill( '*' );
25     cout << setw( 10 ) << dec << x << '\n';
26     cout.setf( ios::left, ios::adjustfield );
27     cout << setw( 10 ) << setfill( '%' ) << x << '\n';
28     cout.setf( ios::internal, ios::adjustfield );
29     cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
30     return 0;
31 }

```

```

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).

```

1 // Fig. 11.25: fig11_25.cpp
2 // Using the ios::showbase flag
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 int main()
7 {
8     int x = 100;
9
10    cout << setiosflags( ios::showbase )
11         << "Printing integers preceded by their base:\n"
12         << x << '\n'
13         << oct << x << '\n'
14         << hex << x << endl;
15    return 0;
16 }

```

```

Printing integers preceded by their base:
100
0144
0x64

```

Fig. 11.25 Using the `ios::showbase` flag.

```

1 // Fig. 11.26: fig11_26.cpp
2 // Displaying floating-point values in system default,
3 // scientific, and fixed formats.
4 #include <iostream.h>
5
6 int main()
7 {
8     double x = .001234567, y = 1.946e9;
9
10    cout << "Displayed in default format:\n"
11         << x << '\t' << y << '\n';
12    cout.setf( ios::scientific, ios::floatfield );
13    cout << "Displayed in scientific format:\n"
14         << x << '\t' << y << '\n';
15    cout.unsetf( ios::scientific );
16    cout << "Displayed in default format after unsetf:\n"
17         << x << '\t' << y << '\n';
18    cout.setf( ios::fixed, ios::floatfield );
19    cout << "Displayed in fixed format:\n"
20         << x << '\t' << y << endl;
21    return 0;
22 }

```

```

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.

```
1 // Fig. 11.27: fig11_27.cpp
2 // Using the ios::uppercase flag
3 #include <iostream.h>
4 #include <iomanip.h>
```

Fig. 11.27 Using the `ios::uppercase` flag (part 1 of 2).

```
5
6 int main()
7 {
8     cout << setiosflags( ios::uppercase )
9         << "Printing uppercase letters in scientific\n"
10        << "notation exponents and hexadecimal values:\n"
11        << 4.345e10 << '\n' << hex << 123456789 << endl;
12     return 0;
13 }
```

```
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).

```

1 // Fig. 11.28: fig11_28.cpp
2 // Demonstrating the flags member function.
3 #include <iostream.h>
4
5 int main()
6 {
7     int i = 1000;
8     double d = 0.0947628;
9
10    cout << "The value of the flags variable is: "
11         << cout.flags()
12         << "\nPrint int and double in original format:\n"
13         << i << '\t' << d << "\n\n";
14    long originalFormat =
15        cout.flags( ios::oct | ios::scientific );

```

Fig. 11.28 Demonstrating the **flags** member function (part 1 of 2).

```

16    cout << "The value of the flags variable is: "
17         << cout.flags()
18         << "\nPrint int and double in a new format\n"
19         << "specified using the flags member function:\n"
20         << i << '\t' << d << "\n\n";
21    cout.flags( originalFormat );
22    cout << "The value of the flags variable is: "
23         << cout.flags()
24         << "\nPrint values in original format again:\n"
25         << i << '\t' << d << endl;
26    return 0;
27 }

```

```

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).

```

1 // Fig. 11.29: fig11_29.cpp
2 // Testing error states.
3 #include <iostream.h>
4
5 int main()
6 {
7     int x;
8     cout << "Before a bad input operation:"
9         << "\ncin.rdstate(): " << cin.rdstate()
10        << "\n    cin.eof(): " << cin.eof()
11        << "\n    cin.fail(): " << cin.fail()
12        << "\n    cin.bad(): " << cin.bad()
13        << "\n    cin.good(): " << cin.good()
14        << "\n\nExpects an integer, but enter a character: ";
15    cin >> x;
16
17    cout << "\nEnter a bad input operation:"
18        << "\ncin.rdstate(): " << cin.rdstate()
19        << "\n    cin.eof(): " << cin.eof()
20        << "\n    cin.fail(): " << cin.fail()
21        << "\n    cin.bad(): " << cin.bad()
22        << "\n    cin.good(): " << cin.good() << "\n\n";
23
24    cin.clear();
25
26    cout << "After cin.clear()"
27        << "\ncin.fail(): " << cin.fail()
28        << "\ncin.good(): " << cin.good() << endl;
29    return 0;
30 }

```

```

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)

- Fig. 12.1** A function template.
- Fig. 12.2** Using template functions.
- Fig. 12.3** Demonstrating class template **Stack**.
- Fig. 12.4** Passing a **Stack** template object to a function template.

```

1  template< class T >
2  void printArray( const T *array, const int count )
3  {
4      for ( int i = 0; i < count; i++ )
5          cout << array[ i ] << " ";
6
7      cout << endl;
8  }

```

Fig. 12.1 A function template.

```

1  // Fig 12.2: fig12_02.cpp
2  // Using template functions
3  #include <iostream.h>
4
5  template< class T >
6  void printArray( const T *array, const int count )
7  {
8      for ( int i = 0; i < count; i++ )
9          cout << array[ i ] << " ";
10
11     cout << endl;
12 }
13
14 int main()
15 {
16     const int aCount = 5, bCount = 7, cCount = 6;
17     int a[ aCount ] = { 1, 2, 3, 4, 5 };
18     double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
19     char c[ cCount ] = "HELLO"; // 6th position for null
20
21     cout << "Array a contains:" << endl;
22     printArray( a, aCount ); // integer template function
23
24     cout << "Array b contains:" << endl;
25     printArray( b, bCount ); // double template function
26
27     cout << "Array c contains:" << endl;
28     printArray( c, cCount ); // character template function
29
30     return 0;
31 }

```

```

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.


```

1 // Fig. 12.3: tstack1.h
2 // Class template Stack
3 #ifndef TSTACK1_H
4 #define TSTACK1_H
5
6 #include <iostream.h>
7
8 template< class T >
9 class Stack {
10 public:
11     Stack( int = 10 );    // default constructor (stack size 10)
12     ~Stack() { delete [] stackPtr; } // destructor
13     bool push( const T& ); // push an element onto the stack
14     bool pop( T& );      // pop an element off the stack

```

Fig. 12.3 Demonstrating class template **Stack** (part 1 of 4).

```

15 private:
16     int size;           // # of elements in the stack
17     int top;           // location of the top element
18     T *stackPtr;      // pointer to the stack
19
20     bool isEmpty() const { return top == -1; } // utility
21     bool isFull() const { return top == size - 1; } // functions
22 };
23
24 // Constructor with default size 10
25 template< class T >
26 Stack< T >::Stack( int s )
27 {
28     size = s > 0 ? s : 10;
29     top = -1;           // Stack is initially empty
30     stackPtr = new T[ size ]; // allocate space for elements
31 }
32
33 // Push an element onto the stack
34 // return true if successful, false otherwise
35 template< class T >
36 bool Stack< T >::push( const T &pushValue )
37 {
38     if ( !isFull() ) {
39         stackPtr[ ++top ] = pushValue; // place item in Stack
40         return true; // push successful
41     }
42     return false; // push unsuccessful
43 }
44
45 // Pop an element off the stack
46 template< class T >
47 bool Stack< T >::pop( T &popValue )
48 {
49     if ( !isEmpty() ) {
50         popValue = stackPtr[ top-- ]; // remove item from Stack
51         return true; // pop successful
52     }
53     return false; // pop unsuccessful
54 }
55
56 #endif

```

Fig. 12.3 Demonstrating class template **Stack** (part 2 of 4).

```
57 // Fig. 12.3: fig12_03.cpp
58 // Test driver for Stack template
59 #include <iostream.h>
60 #include "tstack1.h"
61
62 int main()
63 {
64     Stack< double > doubleStack( 5 );
65     double f = 1.1;
66     cout << "Pushing elements onto doubleStack\n";
67
68     while ( doubleStack.push( f ) ) { // success true returned
69         cout << f << ' ';
70         f += 1.1;
71     }
72
73     cout << "\nStack is full. Cannot push " << f
74         << "\n\nPopping elements from doubleStack\n";
75
76     while ( doubleStack.pop( f ) ) // success true returned
77         cout << f << ' ';
78
79     cout << "\nStack is empty. Cannot pop\n";
80
81     Stack< int > intStack;
82     int i = 1;
83     cout << "\nPushing elements onto intStack\n";
84
85     while ( intStack.push( i ) ) { // success true returned
86         cout << i << ' ';
87         ++i;
88     }
89
90     cout << "\nStack is full. Cannot push " << i
91         << "\n\nPopping elements from intStack\n";
92
93     while ( intStack.pop( i ) ) // success true returned
94         cout << i << ' ';
95
96     cout << "\nStack is empty. Cannot pop\n";
97     return 0;
98 }
```

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).

```

1 // Fig. 12.4: fig12_04.cpp
2 // Test driver for Stack template.
3 // Function main uses a function template to manipulate
4 // objects of type Stack< T >.
5 #include <iostream.h>
6 #include "tstack1.h"
7
8 // Function template to manipulate Stack< T >
9 template< class T >
10 void testStack(
11     Stack< T > &theStack, // reference to the Stack< T >
12     T value, // initial value to be pushed
13     T increment, // increment for subsequent values
14     const char *stackName ) // name of the Stack< T > object
15 {
16     cout << "\nPushing elements onto " << stackName << '\n';
17
18     while ( theStack.push( value ) ) { // success true returned
19         cout << value << ' ';
20         value += increment;
21     }
22
23     cout << "\nStack is full. Cannot push " << value
24         << "\n\nPopping elements from " << stackName << '\n';
25
26     while ( theStack.pop( value ) ) // success true returned
27         cout << value << ' ';
28
29     cout << "\nStack is empty. Cannot pop\n";
30 }

```

Fig. 12.4 Passing a **Stack** template object to a function template (part 1 of 2).

```
31
32 int main()
33 {
34     Stack< double > doubleStack( 5 );
35     Stack< int > intStack;
36
37     testStack( doubleStack, 1.1, 1.1, "doubleStack" );
38     testStack( intStack, 1, 1, "intStack" );
39
40     return 0;
41 }
```

```
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** A simple exception-handling example with divide by zero.
- Fig. 13.2** Rethrowing an exception.
- Fig. 13.3** Demonstration of stack unwinding.
- Fig. 13.4** Demonstrating **new** returning 0 on failure.
- Fig. 13.5** Demonstrating **new** throwing **bad_alloc** on failure.
- Fig. 13.6** Demonstrating **set_new_handler**.
- Fig. 13.7** Demonstrating **auto_ptr**.

```

1 // Fig. 13.1: fig13_01.cpp
2 // A simple exception handling example.
3 // Checking for a divide-by-zero exception.
4 #include <iostream.h>
5
6 // Class DivideByZeroException to be used in exception
7 // handling for throwing an exception on a division by zero.
8 class DivideByZeroException {
9 public:
10    DivideByZeroException()
11        : message( "attempted to divide by zero" ) { }
12    const char *what() const { return message; }
13 private:
14    const char *message;
15 };
16
17 // Definition of function quotient. Demonstrates throwing
18 // an exception when a divide-by-zero exception is encountered.
19 double quotient( int numerator, int denominator )
20 {
21     if ( denominator == 0 )
22         throw DivideByZeroException();
23
24     return static_cast< double > ( numerator ) / denominator;
25 }
26
27 // Driver program
28 int main()
29 {
30     int number1, number2;
31     double result;
32
33     cout << "Enter two integers (end-of-file to end): ";
34
35     while ( cin >> number1 >> number2 ) {
36
37         // the try block wraps the code that may throw an
38         // exception and the code that should not execute
39         // if an exception occurs
40         try {
41             result = quotient( number1, number2 );
42             cout << "The quotient is: " << result << endl;
43         }

```

Fig. 13.1 A simple exception-handling example with divide by zero (part 1 of 2).

```

44     catch ( DivideByZeroException ex ) { // exception handler
45         cout << "Exception occurred: " << ex.what() << '\n';
46     }
47
48     cout << "\nEnter two integers (end-of-file to end): ";
49 }
50
51 cout << endl;
52 return 0; // terminate normally
53 }

```

```

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).

```

1 // Fig. 13.2: fig13_02.cpp
2 // Demonstration of rethrowing an exception.
3 #include <iostream>
4 #include <exception>
5
6 using namespace std;
7
8 void throwException() throw ( exception )
9 {
10 // Throw an exception and immediately catch it.
11 try {
12     cout << "Function throwException\n";
13     throw exception(); // generate exception
14 }
15 catch( exception e )
16 {
17     cout << "Exception handled in function throwException\n";
18     throw; // rethrow exception for further processing
19 }

```

Fig. 13.2 Rethrowing an exception (part 1 of 2).

```

20
21     cout << "This also should not print\n";
22 }
23
24 int main()
25 {
26     try {
27         throwException();
28         cout << "This should not print\n";
29     }
30     catch ( exception e )
31     {
32         cout << "Exception handled in main\n";
33     }
34
35     cout << "Program control continues after catch in main"
36         << endl;
37     return 0;
38 }

```

```

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).

```

1 // Fig. 13.3: fig13_03.cpp
2 // Demonstrating stack unwinding.
3 #include <iostream>
4 #include <stdexcept>
5
6 using namespace std;
7
8 void function3() throw ( runtime_error )
9 {
10     throw runtime_error( "runtime_error in function3" );
11 }
12
13 void function2() throw ( runtime_error )
14 {
15     function3();
16 }
17
18 void function1() throw ( runtime_error )
19 {
20     function2();
21 }
22
23 int main()
24 {
25     try {
26         function1();
27     }
28     catch ( runtime_error e )
29     {
30         cout << "Exception occurred: " << e.what() << endl;
31     }
32
33     return 0;
34 }

```

```
Exception occurred: runtime_error in function3
```

Fig. 13.3 Demonstration of stack unwinding.


```

1 // Fig. 13.4: fig13_04.cpp
2 // Demonstrating new returning 0
3 // when memory is not allocated
4 #include <iostream.h>
5
6 int main()
7 {
8     double *ptr[ 10 ];
9
10    for ( int i = 0; i < 10; i++ ) {
11        ptr[ i ] = new double[ 5000000 ];
12
13        if ( ptr[ i ] == 0 ) { // new failed to allocate memory
14            cout << "Memory allocation failed for ptr[ "
15                << i << " ]\n";
16            break;
17        }
18        else
19            cout << "Allocated 5000000 doubles in ptr[ "
20                << i << " ]\n";
21    }
22
23    return 0;
24 }

```

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).

```

1 // Fig. 13.5: fig13_05.cpp
2 // Demonstrating new throwing bad_alloc
3 // when memory is not allocated
4 #include <iostream>
5 #include <new>
6
7 int main()
8 {
9     double *ptr[ 10 ];
10
11    try {
12        for ( int i = 0; i < 10; i++ ) {
13            ptr[ i ] = new double[ 5000000 ];
14            cout << "Allocated 5000000 doubles in ptr[ "
15                << i << " ]\n";
16        }
17    }
18    catch ( bad_alloc exception ) {
19        cout << "Exception occurred: "
20            << exception.what() << endl;
21    }
22
23    return 0;
24 }

```

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).

```

1 // Fig. 13.6: fig13_06.cpp
2 // Demonstrating set_new_handler
3 #include <iostream.h>
4 #include <new.h>
5 #include <stdlib.h>
6
7 void customNewHandler()
8 {
9     cerr << "customNewHandler was called";
10    abort();
11 }
12
13 int main()
14 {
15     double *ptr[ 10 ];
16     set_new_handler( customNewHandler );
17
18     for ( int i = 0; i < 10; i++ ) {
19         ptr[ i ] = new double[ 5000000 ];
20
21         cout << "Allocated 5000000 doubles in ptr[ "
22              << i << " ]\n";
23     }
24
25     return 0;
26 }

```

```

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`.

```
1 // Fig. 13.7: fig13_07.cpp
2 // Demonstrating auto_ptr
3 #include <iostream>
4 #include <memory>
5
6 using namespace std;
7
8 class Integer {
9 public:
10     Integer( int i = 0 ) : value( i )
11         { cout << "Constructor for Integer " << value << endl; }
12     ~Integer()
13         { cout << "Destructor for Integer " << value << endl; }
14     void setInteger( int i ) { value = i; }
15     int getInteger() const { return value; }
16 private:
17     int value;
18 };
19
20 int main()
21 {
22     cout << "Creating an auto_ptr object that points "
23         << "to an Integer\n";
24
25     auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
26
27     cout << "Using the auto_ptr to manipulate the Integer\n";
28     ptrToInteger->setInteger( 99 );
29     cout << "Integer after setInteger: "
30         << ( *ptrToInteger ).getInteger()
31         << "\nTerminating program" << endl;
32
33     return 0;
34 }
```

```
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`.

Illustrations 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** Creating a sequential file.
- Fig. 14.5** File open modes.
- Fig. 14.6** End-of-file key combinations for various popular computer systems.
- Fig. 14.7** Reading and printing a sequential file.
- Fig. 14.8** Credit inquiry program.
- 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** Creating a random access file sequentially.
- Fig. 14.12** Writing data randomly to a random access file.
- Fig. 14.13** Sample execution of the program in Fig. 14.12.
- Fig. 14.14** Reading a random access file sequentially.
- Fig. 14.15** Bank Account Program.

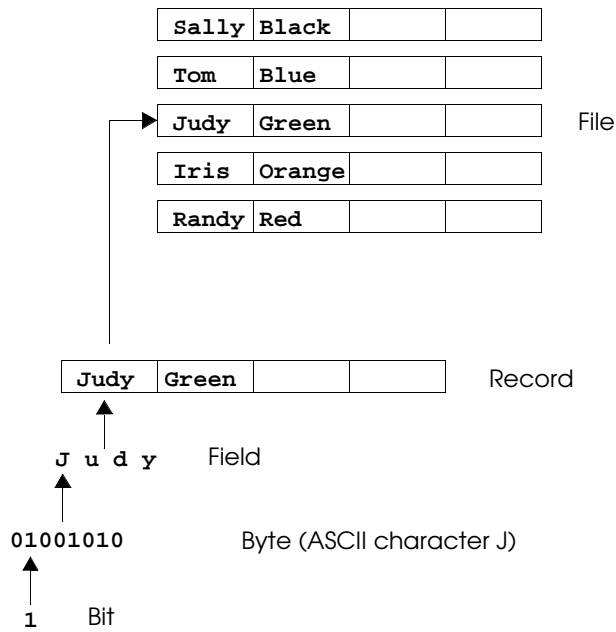


Fig. 14.1 The data hierarchy.

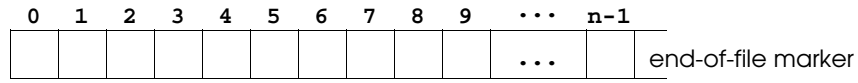


Fig. 14.2 C++'s view of a file of *n* bytes.

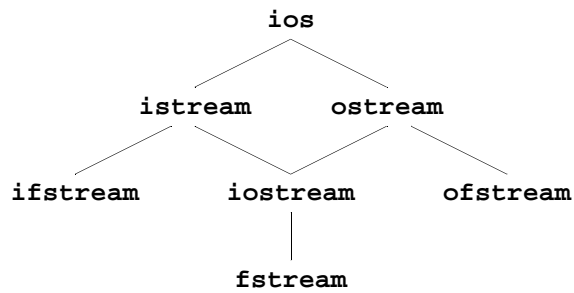


Fig. 14.3 Portion of stream I/O class hierarchy.

```
1 // Fig. 14.4: fig14_04.cpp
2 // Create a sequential file
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     // ofstream constructor opens file
10    ofstream outClientFile( "clients.dat", ios::out );
11
12    if ( !outClientFile ) { // overloaded ! operator
13        cerr << "File could not be opened" << endl;
14        exit( 1 ); // prototype in stdlib.h
15    }
16
17    cout << "Enter the account, name, and balance.\n"
18         << "Enter end-of-file to end input.\n? ";
19
20    int account;
21    char name[ 30 ];
22    float balance;
23
24    while ( cin >> account >> name >> balance ) {
25        outClientFile << account << ' ' << name
26                    << ' ' << balance << '\n';
27        cout << "? ";
28    }
29
30    return 0; // ofstream destructor closes file
31 }
```

```
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
<code>ios::app</code>	Write all output to the end of the file.
<code>ios::ate</code>	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.
<code>ios::in</code>	Open a file for input.
<code>ios::out</code>	Open a file for output.
<code>ios::trunc</code>	Discard the file's contents if it exists (this is also the default action for <code>ios::out</code>)
<code>ios::nocreate</code>	If the file does not exist, the open operation fails.
<code>ios::noreplace</code>	If the file exists, the open operation fails.

Fig. 14.5 File open modes.

Computer system	Keyboard combination
UNIX systems	<code><ctrl> d</code> (on a line by itself)
IBM PC and compatibles	<code><ctrl> z</code>
Macintosh	<code><ctrl> d</code>
VAX (VMS)	<code><ctrl> z</code>

Fig. 14.6 End-of-file key combinations for various popular computer systems.

```

1 // Fig. 14.7: fig14_07.cpp
2 // Reading and printing a sequential file
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <iomanip.h>
6 #include <stdlib.h>
7
8 void outputLine( int, const char *, double );
9
10 int main()
11 {
12     // ifstream constructor opens the file
13     ifstream inClientFile( "clients.dat", ios::in );
14
15     if ( !inClientFile ) {
16         cerr << "File could not be opened\n";
17         exit( 1 );
18     }
19
20     int account;
21     char name[ 30 ];
22     double balance;
23
24     cout << setiosflags( ios::left ) << setw( 10 ) << "Account"

```

```

25         << setw( 13 ) << "Name" << "Balance\n";
26
27     while ( inClientFile >> account >> name >> balance )
28         outputLine( account, name, balance );
29
30     return 0; // ifstream destructor closes the file
31 }

```

Fig. 14.7 Reading and printing a sequential file (part 1 of 2).

```

32
33 void outputLine( int acct, const char *name, double bal )
34 {
35     cout << setiosflags( ios::left ) << setw( 10 ) << acct
36         << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
37         << resetiosflags( ios::left )
38         << setiosflags( ios::fixed | ios::showpoint )
39         << bal << '\n';
40 }

```

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).

```

1 // Fig. 14.8: fig14_08.cpp
2 // Credit inquiry program
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <iomanip.h>
6 #include <stdlib.h>
7
8 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE,
9                 DEBIT_BALANCE, END };
10 int getRequest();
11 bool shouldDisplay( int, double );
12 void outputLine( int, const char *, double );
13
14 int main()
15 {
16     // ifstream constructor opens the file
17     ifstream inClientFile( "clients.dat", ios::in );
18
19     if ( !inClientFile ) {
20         cerr << "File could not be opened" << endl;
21         exit( 1 );
22     }
23
24     int request;
25     int account;
26     char name[ 30 ];
27     double balance;
28
29     cout << "Enter request\n"
30         << " 1 - List accounts with zero balances\n"
31         << " 2 - List accounts with credit balances\n"

```



```

32     << " 3 - List accounts with debit balances\n"
33     << " 4 - End of run";
34     request = getRequest();
35
36     while ( request != END ) {
37
38         switch ( request ) {
39             case ZERO_BALANCE:
40                 cout << "\nAccounts with zero balances:\n";
41                 break;
42             case CREDIT_BALANCE:
43                 cout << "\nAccounts with credit balances:\n";
44                 break;
45             case DEBIT_BALANCE:
46                 cout << "\nAccounts with debit balances:\n";
47                 break;
48         }
49
50         inClientFile >> account >> name >> balance;
51

```

Fig. 14.8 Credit inquiry program (part 1 of 2).

```

52     while ( !inClientFile.eof() ) {
53         if ( shouldDisplay( request, balance ) )
54             outputLine( account, name, balance );
55
56         inClientFile >> account >> name >> balance;
57     }
58
59     inClientFile.clear(); // reset eof for next input
60     inClientFile.seekg( 0 ); // move to beginning of file
61     request = getRequest();
62 }
63
64 cout << "End of run." << endl;
65
66 return 0; // ifstream destructor closes the file
67 }
68
69 int getRequest()
70 {
71     int request;
72
73     do {
74         cout << "\n? ";
75         cin >> request;
76     } while( request < ZERO_BALANCE && request > END );
77
78     return request;
79 }
80
81 bool shouldDisplay( int type, double balance )
82 {
83     if ( type == CREDIT_BALANCE && balance < 0 )
84         return true;
85
86     if ( type == DEBIT_BALANCE && balance > 0 )
87         return true;
88
89     if ( type == ZERO_BALANCE && balance == 0 )
90         return true;
91
92     return false;

```

```

93 }
94
95 void outputLine( int acct, const char *name, double bal )
96 {
97     cout << setiosflags( ios::left ) << setw( 10 ) << acct
98         << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
99         << resetiosflags( ios::left )
100        << setiosflags( ios::fixed | ios::showpoint )
101        << bal << '\n';
102 }

```

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.

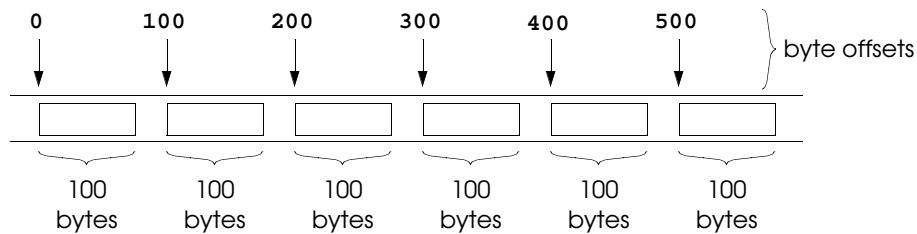


Fig. 14.10 C++'s view of a random access file.

```

1 // Fig. 14.11: clntdata.h
2 // Definition of struct clientData used in
3 // Figs. 14.11, 14.12, 14.14 and 14.15.
4 #ifndef CLNTDATA_H
5 #define CLNTDATA_H
6
7 struct clientData {
8     int accountNumber;
9     char lastName[ 15 ];
10    char firstName[ 10 ];
11    float balance;
12 };
13
14 #endif

```

Fig. 14.11 Creating a random access file sequentially (part 1 of 2).

```

15 // Fig. 14.11: fig14_11.cpp
16 // Creating a randomly accessed file sequentially
17 #include <iostream.h>
18 #include <fstream.h>
19 #include <stdlib.h>
20 #include "clntdata.h"
21
22 int main()
23 {
24     ofstream outCredit( "credit.dat", ios::out );
25
26     if ( !outCredit ) {
27         cerr << "File could not be opened." << endl;
28         exit( 1 );
29     }
30
31     clientData blankClient = { 0, "", "", 0.0 };
32
33     for ( int i = 0; i < 100; i++ )
34         outCredit.write(
35             reinterpret_cast<const char *>( &blankClient ),
36             sizeof( clientData ) );
37     return 0;
38 }

```

Fig. 14.11 Creating a random access file sequentially (part 2 of 2).

```

1 // Fig. 14.12: fig14_12.cpp
2 // Writing to a random access file
3 #include <iostream.h>
4 #include <fstream.h>
5 #include <stdlib.h>
6 #include "clntdata.h"
7
8 int main()
9 {
10     ofstream outCredit( "credit.dat", ios::ate );
11
12     if ( !outCredit ) {
13         cerr << "File could not be opened." << endl;
14         exit( 1 );
15     }
16

```

```

17     cout << "Enter account number "
18         << "(1 to 100, 0 to end input)\n? ";
19
20     clientData client;
21     cin >> client.accountNumber;
22
23     while ( client.accountNumber > 0 &&
24            client.accountNumber <= 100 ) {
25         cout << "Enter lastname, firstname, balance\n? ";
26         cin >> client.lastName >> client.firstName
27             >> client.balance;
28
29         outCredit.seekp( ( client.accountNumber - 1 ) *
30                          sizeof( clientData ) );
31         outCredit.write(
32             reinterpret_cast<const char *>( &client ),
33             sizeof( clientData ) );
34

```

Fig. 14.12 Writing data randomly to a random access file (part 1 of 2).

```

35         cout << "Enter account number\n? ";
36         cin >> client.accountNumber;
37     }
38
39     return 0;
40 }

```

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.

```

1 // Fig. 14.14: fig14_14.cpp
2 // Reading a random access file sequentially
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <fstream.h>
6 #include <stdlib.h>
7 #include "clntdata.h"
8
9 void outputLine( ostream&, const clientData & );
10
11 int main()
12 {
13     ifstream inCredit( "credit.dat", ios::in );
14
15     if ( !inCredit ) {
16         cerr << "File could not be opened." << endl;
17         exit( 1 );
18     }
19
20     cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
21          << setw( 16 ) << "Last Name" << setw( 11 )
22          << "First Name" << resetiosflags( ios::left )
23          << setw( 10 ) << "Balance" << endl;
24
25     clientData client;
26
27     inCredit.read( reinterpret_cast<char *>( &client ),
28                  sizeof( clientData ) );
29

```

Fig. 14.14 Reading a random access file sequentially (part 1 of 2).

```

30     while ( inCredit && !inCredit.eof() ) {
31
32         if ( client.accountNumber != 0 )
33             outputLine( cout, client );
34
35         inCredit.read( reinterpret_cast<char *>( &client ),
36                      sizeof( clientData ) );
37     }
38
39     return 0;
40 }
41
42 void outputLine( ostream &output, const clientData &c )
43 {
44     output << setiosflags( ios::left ) << setw( 10 )
45           << c.accountNumber << setw( 16 ) << c.lastName
46           << setw( 11 ) << c.firstName << setw( 10 )
47           << setprecision( 2 ) << resetiosflags( ios::left )
48           << setiosflags( ios::fixed | ios::showpoint )
49           << c.balance << '\n';

```

50 }

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).

```

1 // Fig. 14.15: fig14_15.cpp
2 // This program reads a random access file sequentially,
3 // updates data already written to the file, creates new
4 // data to be placed in the file, and deletes data
5 // already in the file.
6 #include <iostream.h>
7 #include <fstream.h>
8 #include <iomanip.h>
9 #include <stdlib.h>
10 #include "clntdata.h"
11
12 int enterChoice();
13 void textFile( fstream& );
14 void updateRecord( fstream& );
15 void newRecord( fstream& );
16 void deleteRecord( fstream& );
17 void outputLine( ostream&, const clientData & );

```

Fig. 14.15 Bank account program (part 1 of 5).

```

18 int getAccount( const char * );
19
20 enum Choices { TEXTFILE = 1, UPDATE, NEW, DELETE, END };
21
22 int main()
23 {
24     fstream inOutCredit( "credit.dat", ios::in | ios::out );
25
26     if ( !inOutCredit ) {
27         cerr << "File could not be opened." << endl;
28         exit ( 1 );
29     }
30
31     int choice;
32
33     while ( ( choice = enterChoice() ) != END ) {
34
35         switch ( choice ) {
36             case TEXTFILE:
37                 textFile( inOutCredit );
38                 break;
39             case UPDATE:
40                 updateRecord( inOutCredit );
41                 break;
42             case NEW:
43                 newRecord( inOutCredit );
44                 break;
45             case DELETE:

```

```

46         deleteRecord( inOutCredit );
47         break;
48     default:
49         cerr << "Incorrect choice\n";
50         break;
51     }
52
53     inOutCredit.clear(); // resets end-of-file indicator
54 }
55
56 return 0;
57 }
58
59 // Prompt for and input menu choice
60 int enterChoice()
61 {
62     cout << "\nEnter your choice" << endl
63         << "1 - store a formatted text file of accounts\n"
64         << "    called \"print.txt\" for printing\n"
65         << "2 - update an account\n"
66         << "3 - add a new account\n"
67         << "4 - delete an account\n"
68         << "5 - end program\n? ";

```

Fig. 14.15 Bank account program (part 2 of 5).

```

69
70     int menuChoice;
71     cin >> menuChoice;
72     return menuChoice;
73 }
74
75 // Create formatted text file for printing
76 void textFile( fstream &readFromFile )
77 {
78     ofstream outPrintFile( "print.txt", ios::out );
79
80     if ( !outPrintFile ) {
81         cerr << "File could not be opened." << endl;
82         exit( 1 );
83     }
84
85     outPrintFile << setiosflags( ios::left ) << setw( 10 )
86         << "Account" << setw( 16 ) << "Last Name" << setw( 11 )
87         << "First Name" << resetiosflags( ios::left )
88         << setw( 10 ) << "Balance" << endl;
89     readFromFile.seekg( 0 );
90
91     clientData client;
92     readFromFile.read( reinterpret_cast<char *>( &client ),
93                     sizeof( clientData ) );
94
95     while ( !readFromFile.eof() ) {
96         if ( client.accountNumber != 0 )
97             outputLine( outPrintFile, client );
98
99         readFromFile.read( reinterpret_cast<char *>( &client ),
100                        sizeof( clientData ) );
101     }
102 }
103
104 // Update an account's balance
105 void updateRecord( fstream &updateFile )
106 {

```

```

107     int account = getAccount( "Enter account to update" );
108
109     updateFile.seekg( ( account - 1 ) * sizeof( clientData ) );
110
111     clientData client;
112     updateFile.read( reinterpret_cast<char *>( &client ),
113                    sizeof( clientData ) );
114
115     if ( client.accountNumber != 0 ) {
116         outputLine( cout, client );
117         cout << "\nEnter charge (+) or payment (-): ";
118
119         float transaction;    // charge or payment

```

Fig. 14.15 Bank account program (part 3 of 5).

```

120         cin >> transaction;    // should validate
121         client.balance += transaction;
122         outputLine( cout, client );
123         updateFile.seekp( ( account-1 ) * sizeof( clientData ) );
124         updateFile.write(
125             reinterpret_cast<const char *>( &client ),
126             sizeof( clientData ) );
127     }
128     else
129         cerr << "Account #" << account
130             << " has no information." << endl;
131 }
132
133 // Create and insert new record
134 void newRecord( fstream &insertInFile )
135 {
136     int account = getAccount( "Enter new account number" );
137
138     insertInFile.seekg( ( account-1 ) * sizeof( clientData ) );
139
140     clientData client;
141     insertInFile.read( reinterpret_cast<char *>( &client ),
142                      sizeof( clientData ) );
143
144     if ( client.accountNumber == 0 ) {
145         cout << "Enter lastname, firstname, balance\n? ";
146         cin >> client.lastName >> client.firstName
147             >> client.balance;
148         client.accountNumber = account;
149         insertInFile.seekp( ( account - 1 ) *
150                             sizeof( clientData ) );
151         insertInFile.write(
152             reinterpret_cast<const char *>( &client ),
153             sizeof( clientData ) );
154     }
155     else
156         cerr << "Account #" << account
157             << " already contains information." << endl;
158 }
159
160 // Delete an existing record
161 void deleteRecord( fstream &deleteFromFile )
162 {
163     int account = getAccount( "Enter account to delete" );
164
165     deleteFromFile.seekg( ( account-1 ) * sizeof( clientData ) );
166
167     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** Manipulating a linked list.
- Fig. 15.4** Sample output for the program of Fig. 15.3.
- 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** A simple stack program.
- Fig. 15.10** Sample output from the program of Fig. 15.9.
- Fig. 15.11** A simple stack program using composition.
- Fig. 15.12** Processing a queue.
- 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** Creating and traversing a binary tree.
- Fig. 15.17** Sample output from the program of Fig. 15.16.
- Fig. 15.18** A binary search tree.
- Fig. 15.19** A 15-node binary search tree.
- Fig. 15.20** Simple commands.
- Fig. 15.21** Simple program that determines the sum of two integers.
- Fig. 15.22** Simple program that finds the larger of two integers.
- Fig. 15.23** Calculate the squares of several integers.
- Fig. 15.24** Writing, compiling, and executing a Simple language program.
- Fig. 15.25** SML instructions produced after the compiler's first pass.
- Fig. 15.26** Symbol table for program of Fig. 15.25.
- Fig. 15.27** Unoptimized code from the program of Fig. 15.25.
- Fig. 15.28** Optimized code for the program of Fig. 15.25

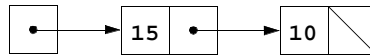


Fig. 15.1 Two self-referential class objects linked together.

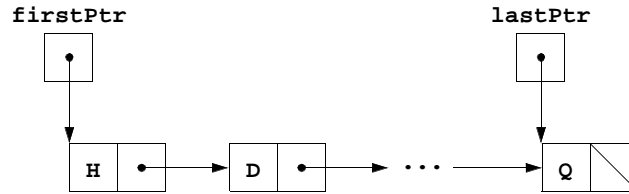


Fig. 15.2 A graphical representation of a list.

```

1 // Fig. 15.3: listnd.h
2 // ListNode template definition
3 #ifndef LISTND_H
4 #define LISTND_H

```

Fig. 15.3 Manipulating a linked list (part 1 of 8).

```

5
6 template< class NODETYPE > class List; // forward declaration
7
8 template<class NODETYPE>
9 class ListNode {
10     friend class List< NODETYPE >; // make List a friend
11 public:
12     ListNode( const NODETYPE & ); // constructor
13     NODETYPE getData() const; // return data in the node
14 private:
15     NODETYPE data; // data
16     ListNode< NODETYPE > *nextPtr; // next node in the list
17 };
18
19 // Constructor
20 template<class NODETYPE>
21 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
22     : data( info ), nextPtr( 0 ) { }
23
24 // Return a copy of the data in the node
25 template< class NODETYPE >
26 NODETYPE ListNode< NODETYPE >::getData() const { return data; }
27
28 #endif

```

Fig. 15.3 Manipulating a linked list (part 2 of 8).

```

29 // Fig. 15.3: list.h
30 // Template List class definition
31 #ifndef LIST_H
32 #define LIST_H
33
34 #include <iostream.h>
35 #include <assert.h>
36 #include "listnd.h"
37
38 template< class NODETYPE >
39 class List {

```

```

40 public:
41     List();           // constructor
42     ~List();         // destructor
43     void insertAtFront( const NODETYPE & );
44     void insertAtBack( const NODETYPE & );
45     bool removeFromFront( NODETYPE & );
46     bool removeFromBack( NODETYPE & );
47     bool isEmpty() const;
48     void print() const;
49 private:
50     ListNode< NODETYPE > *firstPtr; // pointer to first node
51     ListNode< NODETYPE > *lastPtr;  // pointer to last node

```

Fig. 15.3 Manipulating a linked list (part 3 of 8).

```

52
53     // Utility function to allocate a new node
54     ListNode< NODETYPE > *getNode( const NODETYPE & );
55 };
56
57 // Default constructor
58 template< class NODETYPE >
59 List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) { }
60
61 // Destructor
62 template< class NODETYPE >
63 List< NODETYPE >::~~List()
64 {
65     if ( !isEmpty() ) { // List is not empty
66         cout << "Destroying nodes ...\n";
67
68         ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;
69
70         while ( currentPtr != 0 ) { // delete remaining nodes
71             tempPtr = currentPtr;
72             cout << tempPtr->data << '\n';
73             currentPtr = currentPtr->nextPtr;
74             delete tempPtr;
75         }
76     }
77
78     cout << "All nodes destroyed\n\n";
79 }
80
81 // Insert a node at the front of the list
82 template< class NODETYPE >
83 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
84 {
85     ListNode< NODETYPE > *newPtr = getNode( value );
86
87     if ( isEmpty() ) // List is empty
88         firstPtr = lastPtr = newPtr;
89     else { // List is not empty
90         newPtr->nextPtr = firstPtr;
91         firstPtr = newPtr;
92     }
93 }
94
95 // Insert a node at the back of the list
96 template< class NODETYPE >
97 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
98 {
99     ListNode< NODETYPE > *newPtr = getNode( value );
100

```

```

101     if ( isEmpty() ) // List is empty
102         firstPtr = lastPtr = newPtr;

```

Fig. 15.3 Manipulating a linked list (part 4 of 8).

```

103     else { // List is not empty
104         lastPtr->nextPtr = newPtr;
105         lastPtr = newPtr;
106     }
107 }
108
109 // Delete a node from the front of the list
110 template< class NODETYPE >
111 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
112 {
113     if ( isEmpty() ) // List is empty
114         return false; // delete unsuccessful
115     else {
116         ListNode< NODETYPE > *tempPtr = firstPtr;
117
118         if ( firstPtr == lastPtr )
119             firstPtr = lastPtr = 0;
120         else
121             firstPtr = firstPtr->nextPtr;
122
123         value = tempPtr->data; // data being removed
124         delete tempPtr;
125         return true; // delete successful
126     }
127 }
128
129 // Delete a node from the back of the list
130 template< class NODETYPE >
131 bool List< NODETYPE >::removeFromBack( NODETYPE &value )
132 {
133     if ( isEmpty() )
134         return false; // delete unsuccessful
135     else {
136         ListNode< NODETYPE > *tempPtr = lastPtr;
137
138         if ( firstPtr == lastPtr )
139             firstPtr = lastPtr = 0;
140         else {
141             ListNode< NODETYPE > *currentPtr = firstPtr;
142
143             while ( currentPtr->nextPtr != lastPtr )
144                 currentPtr = currentPtr->nextPtr;
145
146             lastPtr = currentPtr;
147             currentPtr->nextPtr = 0;
148         }
149
150         value = tempPtr->data;
151         delete tempPtr;

```

Fig. 15.3 Manipulating a linked list (part 5 of 8).

```

152         return true; // delete successful
153     }
154 }
155
156 // Is the List empty?

```

```

157 template< class NODETYPE >
158 bool List< NODETYPE >::isEmpty() const
159     { return firstPtr == 0; }
160
161 // Return a pointer to a newly allocated node
162 template< class NODETYPE >
163 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
164     const NODETYPE &value )
165 {
166     ListNode< NODETYPE > *ptr =
167         new ListNode< NODETYPE >( value );
168     assert( ptr != 0 );
169     return ptr;
170 }
171
172 // Display the contents of the List
173 template< class NODETYPE >
174 void List< NODETYPE >::print() const
175 {
176     if ( isEmpty() ) {
177         cout << "The list is empty\n\n";
178         return;
179     }
180
181     ListNode< NODETYPE > *currentPtr = firstPtr;
182
183     cout << "The list is: ";
184
185     while ( currentPtr != 0 ) {
186         cout << currentPtr->data << ' ';
187         currentPtr = currentPtr->nextPtr;
188     }
189
190     cout << "\n\n";
191 }
192
193 #endif

```

Fig. 15.3 Manipulating a linked list (part 6 of 8).

```

194 // Fig. 15.3: fig15_03.cpp
195 // List class test
196 #include <iostream.h>
197 #include "list.h"
198
199 // Function to test an integer List
200 template< class T >
201 void testList( List< T > &listObject, const char *type )
202 {
203     cout << "Testing a List of " << type << " values\n";
204
205     instructions();
206     int choice;
207     T value;
208
209     do {
210         cout << "? ";
211         cin >> choice;
212
213         switch ( choice ) {
214             case 1:
215                 cout << "Enter " << type << ": ";
216                 cin >> value;
217                 listObject.insertAtFront( value );

```

```

218         listObject.print();
219         break;
220     case 2:
221         cout << "Enter " << type << ": ";
222         cin >> value;
223         listObject.insertAtBack( value );
224         listObject.print();
225         break;
226     case 3:
227         if ( listObject.removeFromFront( value ) )
228             cout << value << " removed from list\n";
229
230         listObject.print();
231         break;
232     case 4:
233         if ( listObject.removeFromBack( value ) )
234             cout << value << " removed from list\n";
235
236         listObject.print();
237         break;
238     }
239 } while ( choice != 5 );
240
241 cout << "End list test\n\n";
242 }
243

```

Fig. 15.3 Manipulating a linked list (part 7 of 8).

```

244 void instructions()
245 {
246     cout << "Enter one of the following:\n"
247         << " 1 to insert at beginning of list\n"
248         << " 2 to insert at end of list\n"
249         << " 3 to delete from beginning of list\n"
250         << " 4 to delete from end of list\n"
251         << " 5 to end list processing\n";
252 }
253
254 int main()
255 {
256     List< int > integerList;
257     testList( integerList, "integer" ); // test integerList
258
259     List< float > floatList;
260     testList( floatList, "float" ); // test integerList
261
262     return 0;
263 }

```

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).

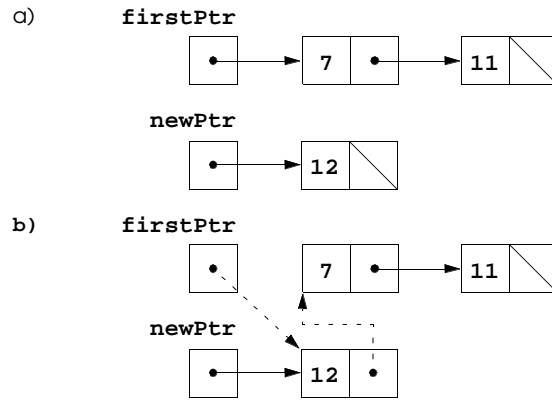


Fig. 15.5 The `insertAtFront` operation.

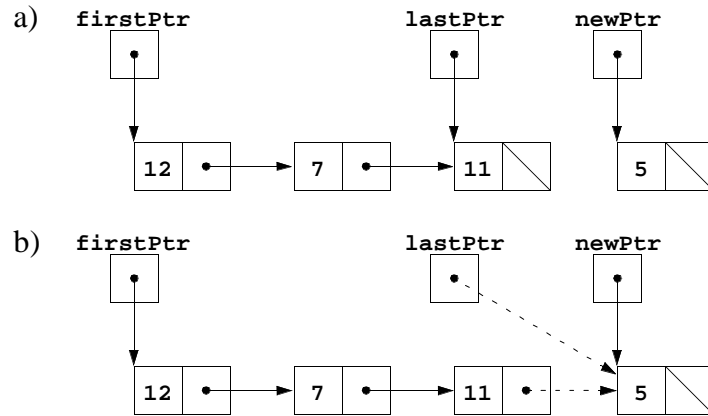


Fig. 15.6 A graphical representation of the `insertAtBack` operation.

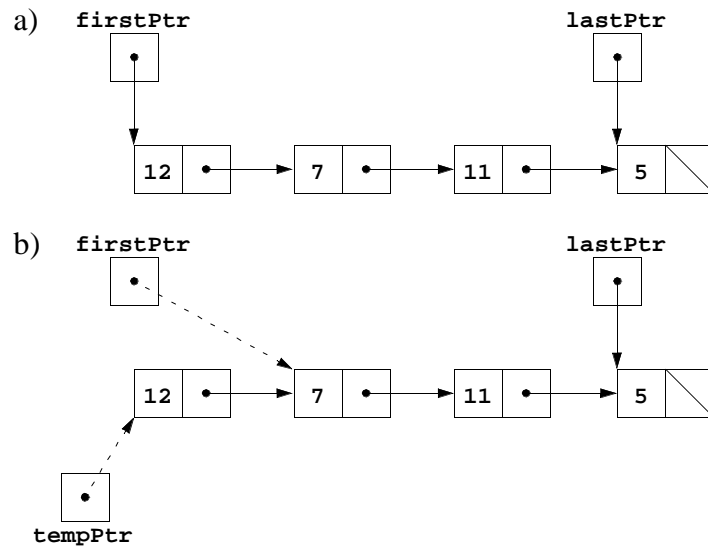


Fig. 15.7 A graphical representation of the `removeFromFront` operation.

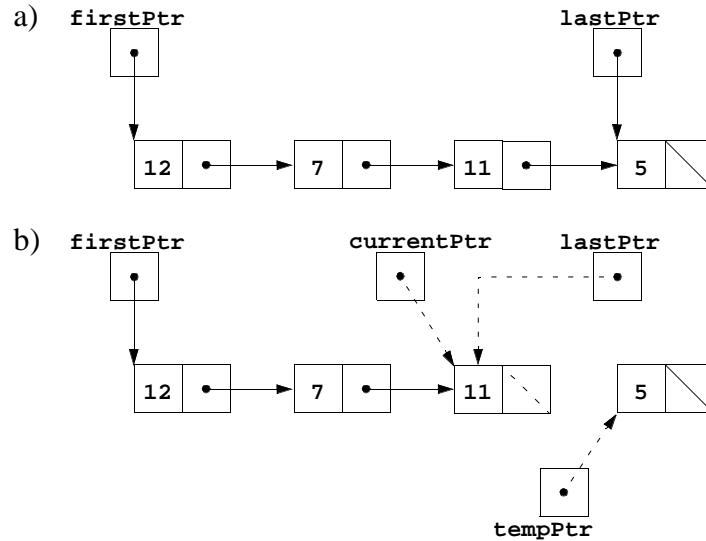


Fig. 15.8 A graphical representation of the `removeFromBack` operation.

```

1 // Fig. 15.9: stack.h
2 // Stack class template definition
3 // Derived from class List
4 #ifndef STACK_H
5 #define STACK_H
6
7 #include "list.h"
8
9 template< class STACKTYPE >
10 class Stack : private List< STACKTYPE > {

```

Fig. 15.9 A simple stack program (part 1 of 3).

```

11 public:
12     void push( const STACKTYPE &d ) { insertAtFront( d ); }
13     bool pop( STACKTYPE &d ) { return removeFromFront( d ); }
14     bool isEmpty() const { return isEmpty(); }
15     void printStack() const { print(); }
16 };
17
18 #endif

```

Fig. 15.9 A simple stack program (part 2 of 3).

```

19 // Fig. 15.9: fig15_09.cpp
20 // Driver to test the template Stack class
21 #include <iostream.h>
22 #include "stack.h"
23
24 int main()
25 {
26     Stack< int > intStack;
27     int popInteger;
28     cout << "processing an integer Stack" << endl;
29
30     for ( int i = 0; i < 4; i++ ) {
31         intStack.push( i );

```

```
32     intStack.printStack();
33 }
34
35 while ( !intStack.isStackEmpty() ) {
36     intStack.pop( popInteger );
37     cout << popInteger << " popped from stack" << endl;
38     intStack.printStack();
39 }
40
41 Stack< double > doubleStack;
42 double val = 1.1, popdouble;
43 cout << "processing a double Stack" << endl;
44
45 for ( i = 0; i < 4; i++ ) {
46     doubleStack.push( val );
47     doubleStack.printStack();
48     val += 1.1;
49 }
50
51 while ( !doubleStack.isStackEmpty() ) {
52     doubleStack.pop( popdouble );
53     cout << popdouble << " popped from stack" << endl;
54     doubleStack.printStack();
55 }
56 return 0;
57 }
```

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.

```

1 // Fig. 15.11: stack_c.h
2 // Definition of Stack class composed of List object
3 #ifndef STACK_C
4 #define STACK_C
5 #include "list.h"
6
7 template< class STACKTYPE >
8 class Stack {
9 public:
10 // no constructor; List constructor does initialization

```

```

11     void push( const STACKTYPE &d ) { s.insertAtFront( d ); }
12     bool pop( STACKTYPE &d ) { return s.removeFromFront( d ); }
13     bool isEmpty() const { return s.isEmpty(); }
14     void printStack() const { s.print(); }
15 private:
16     List< STACKTYPE > s;
17 };
18
19 #endif

```

Fig. 15.11 A simple stack program using composition.

```

1 // Fig. 15.12: queue.h
2 // Queue class template definition
3 // Derived from class List
4 #ifndef QUEUE_H
5 #define QUEUE_H
6
7 #include "list.h"
8
9 template< class QUEUETYPE >
10 class Queue: private List< QUEUETYPE > {
11 public:
12     void enqueue( const QUEUETYPE &d ) { insertAtBack( d ); }
13     bool dequeue( QUEUETYPE &d )
14         { return removeFromFront( d ); }
15     bool isEmpty() const { return isEmpty(); }
16     void printQueue() const { print(); }
17 };
18
19 #endif

```

Fig. 15.12 Processing a queue (part 1 of 2).

```

20 // Fig. 15.12: fig15_12.cpp
21 // Driver to test the template Queue class
22 #include <iostream.h>
23 #include "queue.h"
24
25 int main()
26 {
27     Queue< int > intQueue;
28     int dequeueInteger;
29     cout << "processing an integer Queue" << endl;
30
31     for ( int i = 0; i < 4; i++ ) {
32         intQueue.enqueue( i );
33         intQueue.printQueue();
34     }
35
36     while ( !intQueue.isEmpty() ) {
37         intQueue.dequeue( dequeueInteger );
38         cout << dequeueInteger << " dequeued" << endl;
39         intQueue.printQueue();
40     }
41
42     Queue< double > doubleQueue;
43     double val = 1.1, dequeuedouble;
44
45     cout << "processing a double Queue" << endl;
46

```

```
47     for ( i = 0; i < 4; i++ ) {
48         doubleQueue.enqueue( val );
49         doubleQueue.printQueue();
50         val += 1.1;
51     }
52
53     while ( !doubleQueue.isEmpty() ) {
54         doubleQueue.dequeue( dequeuedouble );
55         cout << dequeuedouble << " dequeued" << endl;
56         doubleQueue.printQueue();
57     }
58
59     return 0;
60 }
```

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.

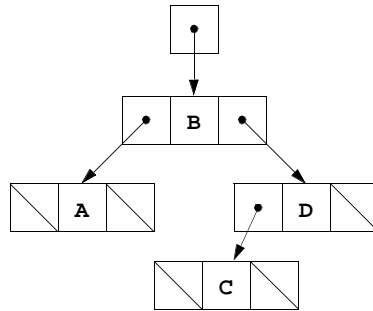


Fig. 15.14 A graphical representation of a binary tree.

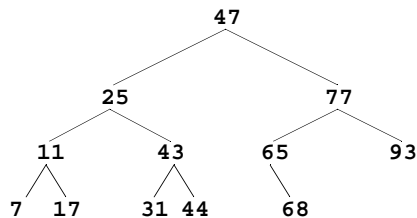


Fig. 15.15 A binary search tree.

```

1 // Fig. 15.16: treenode.h
2 // Definition of class TreeNode
3 #ifndef TREENODE_H
4 #define TREENODE_H
5
6 template< class NODETYPE > class Tree; // forward declaration
7
8 template< class NODETYPE >
9 class TreeNode {
10     friend class Tree< NODETYPE >;
11 public:
12     TreeNode( const NODETYPE &d )
13         : leftPtr( 0 ), data( d ), rightPtr( 0 ) { }
14     NODETYPE getData() const { return data; }
15 private:
16     TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
17     NODETYPE data;
18     TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
19 };
20
21 #endif

```

Fig. 15.16 Creating and traversing a binary tree (part 1 of 6).

```

22 // Fig. 15.16: fig15_16.cpp
23 // Definition of template class Tree
24 #ifndef TREE_H
25 #define TREE_H
26
27 #include <iostream.h>
28 #include <assert.h>
29 #include "treenode.h"

```



```

30
31 template< class NODETYPE >
32 class Tree {
33 public:
34     Tree();
35     void insertNode( const NODETYPE & );
36     void preOrderTraversal() const;
37     void inOrderTraversal() const;
38     void postOrderTraversal() const;
39 private:
40     TreeNode< NODETYPE > *rootPtr;
41
42     // utility functions
43     void insertNodeHelper(
44         TreeNode< NODETYPE > **, const NODETYPE & );
45     void preOrderHelper( TreeNode< NODETYPE > * ) const;
46     void inOrderHelper( TreeNode< NODETYPE > * ) const;
47     void postOrderHelper( TreeNode< NODETYPE > * ) const;
48 };
49
50 template< class NODETYPE >
51 Tree< NODETYPE >::Tree() { rootPtr = 0; }
52
53 template< class NODETYPE >
54 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
55     { insertNodeHelper( &rootPtr, value ); }

```

Fig. 15.16 Creating and traversing a binary tree (part 2 of 6).

```

56
57 // This function receives a pointer to a pointer so the
58 // pointer can be modified.
59 template< class NODETYPE >
60 void Tree< NODETYPE >::insertNodeHelper(
61     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
62 {
63     if ( *ptr == 0 ) { // tree is empty
64         *ptr = new TreeNode< NODETYPE >( value );
65         assert( *ptr != 0 );
66     }
67     else // tree is not empty
68         if ( value < ( *ptr )->data )
69             insertNodeHelper( &( ( *ptr )->leftPtr ), value );
70         else
71             if ( value > ( *ptr )->data )
72                 insertNodeHelper( &( ( *ptr )->rightPtr ), value );
73             else
74                 cout << value << " dup" << endl;
75 }
76
77 template< class NODETYPE >
78 void Tree< NODETYPE >::preOrderTraversal() const
79     { preOrderHelper( rootPtr ); }
80
81 template< class NODETYPE >
82 void Tree< NODETYPE >::preOrderHelper(
83     TreeNode< NODETYPE > *ptr ) const
84 {
85     if ( ptr != 0 ) {
86         cout << ptr->data << ' ';
87         preOrderHelper( ptr->leftPtr );
88         preOrderHelper( ptr->rightPtr );
89     }
90 }

```

```

91
92 template< class NODETYPE >
93 void Tree< NODETYPE >::inOrderTraversal() const
94     { inOrderHelper( rootPtr ); }
95
96 template< class NODETYPE >
97 void Tree< NODETYPE >::inOrderHelper(
98     TreeNode< NODETYPE > *ptr ) const
99     {
100     if ( ptr != 0 ) {
101         inOrderHelper( ptr->leftPtr );
102         cout << ptr->data << ' ';
103         inOrderHelper( ptr->rightPtr );
104     }
105 }

```

Fig. 15.16 Creating and traversing a binary tree (part 3 of 6).

```

106
107 template< class NODETYPE >
108 void Tree< NODETYPE >::postOrderTraversal() const
109     { postOrderHelper( rootPtr ); }
110
111 template< class NODETYPE >
112 void Tree< NODETYPE >::postOrderHelper(
113     TreeNode< NODETYPE > *ptr ) const
114     {
115     if ( ptr != 0 ) {
116         postOrderHelper( ptr->leftPtr );
117         postOrderHelper( ptr->rightPtr );
118         cout << ptr->data << ' ';
119     }
120 }
121
122 #endif

```

Fig. 15.16 Creating and traversing a binary tree (part 4 of 6).

```

123 // Fig. 15.16: fig15_16.cpp
124 // Driver to test class Tree
125 #include <iostream.h>
126 #include <iomanip.h>
127 #include "tree.h"
128
129 int main()
130 {
131     Tree< int > intTree;
132     int intVal;
133
134     cout << "Enter 10 integer values:\n";
135     for( int i = 0; i < 10; i++ ) {
136         cin >> intVal;
137         intTree.insertNode( intVal );
138     }
139
140     cout << "\nPreorder traversal\n";
141     intTree.preOrderTraversal();
142
143     cout << "\nInorder traversal\n";
144     intTree.inOrderTraversal();
145
146     cout << "\nPostorder traversal\n";

```

```

147     intTree.postOrderTraversal();
148
149     Tree< double > doubleTree;
150     double doubleVal;
151

```

Fig. 15.16 Creating and traversing a binary tree (part 5 of 6).

```

152     cout << "\n\nEnter 10 double values:\n"
153           << setiosflags( ios::fixed | ios::showpoint )
154           << setprecision( 1 );
155     for ( i = 0; i < 10; i++ ) {
156         cin >> doubleVal;
157         doubleTree.insertNode( doubleVal );
158     }
159
160     cout << "\nPreorder traversal\n";
161     doubleTree.preOrderTraversal();
162
163     cout << "\nInorder traversal\n";
164     doubleTree.inOrderTraversal();
165
166     cout << "\nPostorder traversal\n";
167     doubleTree.postOrderTraversal();
168
169     return 0;
170 }

```

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.

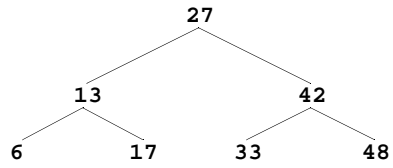


Fig. 15.18 A binary search tree.

Illustrations List (Main Page)

- Fig. 16.1** A possible storage alignment for a variable of type Example showing an undefined area in memory.
- Fig. 16.2** High-performance card shuffling and dealing simulation.
- Fig. 16.3** Output for the high-performance card shuffling and dealing simulation.
- Fig. 16.4** The bitwise operators.
- Fig. 16.5** Printing an unsigned integer in bits.
- Fig. 16.6** Results of combining two bits with the bitwise AND operator &.
- Fig. 16.7** Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and bitwise complement operators.
- Fig. 16.8** Output for the program of Fig. 16.7.
- Fig. 16.9** Results of combining two bits with the bitwise inclusive OR operator |.
- Fig. 16.10** Results of combining two bits with the bitwise exclusive OR operator ^.
- Fig. 16.11** Using the bitwise shift operators.
- Fig. 16.12** The bitwise assignment operators.
- Fig. 16.13** Operator precedence and associativity.
- Fig. 16.14** Using bit fields to store a deck of cards.
- Fig. 16.15** Output of the program in Fig. 16.14.
- Fig. 16.16** Summary of the character handling library functions.
- Fig. 16.17** Using `isdigit`, `isalpha`, `isalnum`, and `isxdigit`.
- Fig. 16.18** Using `islower`, `isupper`, `tolower`, and `toupper`.
- Fig. 16.19** Using `isspace`, `isctrl`, `ispunct`, `isprint`, and `isgraph`.
- Fig. 16.20** Summary of the string conversion functions of the general utilities library.
- Fig. 16.21** Using `atof`.
- Fig. 16.22** Using `atoi`.
- Fig. 16.23** Using `atol`.
- Fig. 16.24** Using `strtod`.
- Fig. 16.25** Using `strtoul`.
- Fig. 16.26** Using `strtoul`.
- Fig. 16.27** Search functions of the string handling library.
- Fig. 16.28** Using `strchr`.
- Fig. 16.29** Using `strcspn`.
- Fig. 16.30** Using `strpbrk`.
- Fig. 16.31** Using `strrchr`.
- Fig. 16.32** Using `strspn`.
- Fig. 16.33** Using `strstr`.
- Fig. 16.34** The memory functions of the string handling library.
- Fig. 16.35** Using `memcpy`.
- Fig. 16.36** Using `memmove`.
- Fig. 16.37** Using `memcmp`.
- Fig. 16.38** Using `memchr`.
- Fig. 16.39** Using `memset`.
- Fig. 16.40** Another string manipulation function of the string handling library.
- Fig. 16.41** Using `strerror`.

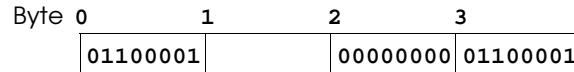


Fig. 16.1 A possible storage alignment for a variable of type **Example** showing an undefined area in memory.

```

1 // Fig. 16.2: fig16_02.cpp
2 // Card shuffling and dealing program using structures
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdlib.h>
6 #include <time.h>
7
8 struct Card {
9     char *face;
10    char *suit;
11 };
12
13 void fillDeck( Card *, char *[], char *[] );
14 void shuffle( Card * );
15 void deal( Card * );
16
17 int main()
18 {
19     Card deck[ 52 ];
20     char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
21                   "Six", "Seven", "Eight", "Nine", "Ten",
22                   "Jack", "Queen", "King" };
23     char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
24
25     srand( time( 0 ) );           // randomize

```

Fig. 16.2 High-performance card shuffling and dealing simulation (part 1 of 2).

```

26     fillDeck( deck, face, suit );
27     shuffle( deck );
28     deal( deck );
29     return 0;
30 }
31
32 void fillDeck( Card *wDeck, char *wFace[], char *wSuit[] )
33 {
34     for ( int i = 0; i < 52; i++ ) {
35         wDeck[ i ].face = wFace[ i % 13 ];
36         wDeck[ i ].suit = wSuit[ i / 13 ];
37     }
38 }
39
40 void shuffle( Card *wDeck )
41 {
42     for ( int i = 0; i < 52; i++ ) {
43         int j = rand() % 52;
44         Card temp = wDeck[ i ];
45         wDeck[ i ] = wDeck[ j ];
46         wDeck[ j ] = temp;
47     }
48 }
49
50 void deal( Card *wDeck )
51 {

```

```

52     for ( int i = 0; i < 52; i++ )
53         cout << setiosflags( ios::right )
54             << setw( 5 ) << wDeck[ i ].face << " of "
55             << setiosflags( ios::left )
56             << setw( 8 ) << wDeck[ i ].suit
57             << ( ( i + 1 ) % 2 ? '\t' : '\n' );
58     }

```

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.

```

1 // Fig. 16.5: fig16_05.cpp
2 // Printing an unsigned integer in bits
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void displayBits( unsigned );
7
8 int main()
9 {
10     unsigned x;
11
12     cout << "Enter an unsigned integer: ";
13     cin >> x;
14     displayBits( x );
15     return 0;
16 }
17
18 void displayBits( unsigned value )
19 {
20     unsigned c, displayMask = 1 << 15;
21
22     cout << setw( 7 ) << value << " = ";
23
24     for ( c = 1; c <= 16; c++ ) {
25         cout << ( value & displayMask ? '1' : '0' );
26         value <<= 1;
27
28         if ( c % 8 == 0 )
29             cout << ' ';
30     }
31     cout << endl;
32 }
33

```



```
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 (&).

```
1 // Fig. 16.7: fig16_07.cpp
2 // Using the bitwise AND, bitwise inclusive OR, bitwise
3 // exclusive OR, and bitwise complement operators.
4 #include <iostream.h>
5 #include <iomanip.h>
6
7 void displayBits( unsigned );
8
9 int main()
10 {
11     unsigned number1, number2, mask, setBits;
12
13     number1 = 65535;
14     mask = 1;
```

Fig. 16.7 Using the bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and bitwise complement operators (part 1 of 2).

```
15     cout << "The result of combining the following\n";
16     displayBits( number1 );
17     displayBits( mask );
18     cout << "using the bitwise AND operator & is\n";
19     displayBits( number1 & mask );
20
21     number1 = 15;
22     setBits = 241;
23     cout << "\nThe result of combining the following\n";
24     displayBits( number1 );
25     displayBits( setBits );
26     cout << "using the bitwise inclusive OR operator | is\n";
27     displayBits( number1 | setBits );
28
29     number1 = 139;
30     number2 = 199;
31     cout << "\nThe result of combining the following\n";
32     displayBits( number1 );
33     displayBits( number2 );
34     cout << "using the bitwise exclusive OR operator ^ is\n";
35     displayBits( number1 ^ number2 );
36
```

```

37     number1 = 21845;
38     cout << "\nThe one's complement of\n";
39     displayBits( number1 );
40     cout << "is" << endl;
41     displayBits( ~number1 );
42
43     return 0;
44 }
45
46 void displayBits( unsigned value )
47 {
48     unsigned c, displayMask = 1 << 15;
49
50     cout << setw( 7 ) << value << " = ";
51
52     for ( c = 1; c <= 16; c++ ) {
53         cout << ( value & displayMask ? '1' : '0' );
54         value <<= 1;
55
56         if ( c % 8 == 0 )
57             cout << ' ';
58     }
59     cout << endl;
60 }
61 }

```

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 (^).

```

1 // Fig. 16.11: fig16_11.cpp
2 // Using the bitwise shift operators
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 void displayBits( unsigned );
7
8 int main()
9 {
10     unsigned number1 = 960;
11
12     cout << "The result of left shifting\n";
13     displayBits( number1 );
14     cout << "8 bit positions using the left "
15         << "shift operator is\n";

```

Fig. 16.11 Using the bitwise shift operators (part 1 of 2).

```

16     displayBits( number1 << 8 );
17     cout << "\nThe result of right shifting\n";
18     displayBits( number1 );
19     cout << "8 bit positions using the right "
20         << "shift operator is\n";
21     displayBits( number1 >> 8 );
22     return 0;
23 }
24
25 void displayBits( unsigned value )
26 {
27     unsigned c, displayMask = 1 << 15;
28
29     cout << setw( 7 ) << value << " = ";
30
31     for ( c = 1; c <= 16; c++ ) {
32         cout << ( value & displayMask ? '1' : '0' );

```

```

33     value <<= 1;
34
35     if ( c % 8 == 0 )
36         cout << ' ';
37 }
38
39 cout << endl;
40 }

```

```

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		right to left unary
* & new		
* / %	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).

Operators	Associa- tivity	Type
&&	left to right	logical AND
	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %= &= = ^= << >> = =	right to left	assignment
,	left to right	comma

Fig. 16.13 Operator precedence and associativity (part 2 of 2).

```

1 // Fig. 16.14: fig16_14.cpp
2 // Example using a bit field
3 #include <iostream.h>
4 #include <iomanip.h>
5
6 struct BitCard {
7     unsigned face : 4;
8     unsigned suit : 2;
9     unsigned color : 1;
10 };
11
12 void fillDeck( BitCard * );
13 void deal( BitCard * );
14
15 int main()
16 {
17     BitCard deck[ 52 ];
18
19     fillDeck( deck );
20     deal( deck );
21     return 0;
22 }
23
24 void fillDeck( BitCard *wDeck )
25 {
26     for ( int i = 0; i <= 51; i++ ) {
27         wDeck[ i ].face = i % 13;
28         wDeck[ i ].suit = i / 13;
29         wDeck[ i ].color = i / 26;
30     }
31 }
32
33 // Output cards in two column format. Cards 0-25 subscripted
34 // with k1 (column 1). Cards 26-51 subscripted k2 in (column 2.)
35 void deal( BitCard *wDeck )
36 {
37     for ( int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
38         cout << "Card:" << setw( 3 ) << wDeck[ k1 ].face
39              << " Suit:" << setw( 2 ) << wDeck[ k1 ].suit
40              << " Color:" << setw( 2 ) << wDeck[ k1 ].color
41              << " " << "Card:" << setw( 3 ) << wDeck[ k2 ].face
42              << " Suit:" << setw( 2 ) << wDeck[ k2 ].suit
43              << " Color:" << setw( 2 ) << wDeck[ k2 ].color

```

```

44         << endl;
45     }
46 }

```

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
<code>int isdigit(int c)</code>	Returns true if c is a digit, and false otherwise.
<code>int isalpha(int c)</code>	Returns true if c is a letter, and false otherwise.
<code>int isalnum(int c)</code>	Returns true if c is a digit or a letter, and false otherwise.
<code>int isxdigit(int c)</code>	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.)
<code>int islower(int c)</code>	Returns true if c is a lowercase letter, and false otherwise.
<code>int isupper(int c)</code>	Returns true if c is an uppercase letter; false otherwise.
<code>int tolower(int c)</code>	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
<code>int toupper(int c)</code>	If <code>c</code> is a lowercase letter, <code>toupper</code> returns <code>c</code> as an uppercase letter. Otherwise, <code>toupper</code> returns the argument unchanged.
<code>int isspace(int c)</code>	Returns <code>true</code> if <code>c</code> is a white-space character—newline (<code>'\n'</code>), space (<code>' '</code>), form feed (<code>'\f'</code>), carriage return (<code>'\r'</code>), horizontal tab (<code>'\t'</code>), or vertical tab (<code>'\v'</code>)—and <code>false</code> otherwise
<code>int iscntrl(int c)</code>	Returns <code>true</code> if <code>c</code> is a control character, and <code>false</code> otherwise.
<code>int ispunct(int c)</code>	Returns <code>true</code> if <code>c</code> is a printing character other than a space, a digit, or a letter, and <code>false</code> otherwise.
<code>int isprint(int c)</code>	Returns <code>true</code> value if <code>c</code> is a printing character including space (<code>' '</code>), and <code>false</code> otherwise.
<code>int isgraph(int c)</code>	Returns <code>true</code> if <code>c</code> is a printing character other than space (<code>' '</code>), and <code>false</code> otherwise.

Fig. 16.16 Summary of the character handling library functions (part 2 of 2).

```

1 // Fig. 16.17: fig16_17.cpp
2 // Using functions isdigit, isalpha, isalnum, and isxdigit
3 #include <iostream.h>
4 #include <ctype.h>
5
6 int main()
7 {
8     cout << "According to isdigit:\n"
9         << ( isdigit( '8' ) ? "8 is a" : "8 is not a" )
10        << " digit\n"
11        << ( isdigit( '#' ) ? "# is a" : "# is not a" )
12        << " digit\n";
13     cout << "\nAccording to isalpha:\n"
14         << ( isalpha( 'A' ) ? "A is a" : "A is not a" )
15         << " letter\n"

```

Fig. 16.17 Using `isdigit`, `isalpha`, `isalnum`, and `isxdigit` (part 1 of 2).

```

16        << ( isalpha( 'b' ) ? "b is a" : "b is not a" )
17        << " letter\n"
18        << ( isalpha( '&' ) ? "& is a" : "& is not a" )
19        << " letter\n"
20        << ( isalpha( '4' ) ? "4 is a" : "4 is not a" )
21        << " letter\n";
22     cout << "\nAccording to isalnum:\n"
23         << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
24         << " digit or a letter\n"
25         << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
26         << " digit or a letter\n"
27         << ( isalnum( '#' ) ? "# is a" : "# is not a" )
28         << " digit or a letter\n";
29     cout << "\nAccording to isxdigit:\n"
30         << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
31         << " hexadecimal digit\n"
32         << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
33         << " hexadecimal digit\n"

```

```

34     << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
35     << " hexadecimal digit\n"
36     << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
37     << " hexadecimal digit\n"
38     << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
39     << " hexadecimal digit" << endl;
40     return 0;
41 }

```

```

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).

```

1 // Fig. 16.18: fig16_18.cpp
2 // Using functions islower, isupper, tolower, toupper
3 #include <iostream.h>
4 #include <ctype.h>
5
6 int main()
7 {
8     cout << "According to islower:\n"
9         << ( islower( 'p' ) ? "p is a" : "p is not a" )
10        << " lowercase letter\n"
11        << ( islower( 'P' ) ? "P is a" : "P is not a" )
12        << " lowercase letter\n"
13        << ( islower( '5' ) ? "5 is a" : "5 is not a" )
14        << " lowercase letter\n"
15        << ( islower( '!' ) ? "! is a" : "! is not a" )
16        << " lowercase letter\n";
17     cout << "\nAccording to isupper:\n"
18         << ( isupper( 'D' ) ? "D is an" : "D is not an" )
19         << " uppercase letter\n"
20         << ( isupper( 'd' ) ? "d is an" : "d is not an" )
21         << " uppercase letter\n"
22         << ( isupper( '8' ) ? "8 is an" : "8 is not an" )
23         << " uppercase letter\n"
24         << ( isupper('$') ? "$ is an" : "$ is not an" )
25         << " uppercase letter\n";
26     cout << "\nu converted to uppercase is "
27         << ( char ) toupper( 'u' )
28         << "\n7 converted to uppercase is "

```



```
29     << ( char ) toupper( '7' )
```

Fig. 16.18 Using `islower`, `isupper`, `tolower`, and `toupper` (part 1 of 2).

```
30     << "\n$ converted to uppercase is "
31     << ( char ) toupper( '$' )
32     << "\nL converted to lowercase is "
33     << ( char ) tolower( 'L' ) << endl;
34
35     return 0;
36 }
```

```
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).

```
1 // Fig. 16.19: fig16_19.cpp
2 // Using functions isspace, iscntrl, ispunct, isprint, isgraph
3 #include <iostream.h>
4 #include <ctype.h>
5
6 int main()
7 {
```

Fig. 16.19 Using `isspace`, `iscntrl`, `ispunct`, `isprint`, and `isgraph` (part 1 of 3).

```
8     cout << "According to isspace:\nNewline "
9         << ( isspace( '\n' ) ? "is a" : "is not a" )
10        << " whitespace character\nHorizontal tab "
11        << ( isspace( '\t' ) ? "is a" : "is not a" )
12        << " whitespace character\n"
13        << ( isspace( '%' ) ? "% is a" : "% is not a" )
14        << " whitespace character\n";
15     cout << "\nAccording to iscntrl:\nNewline "
16         << ( iscntrl( '\n' ) ? "is a" : "is not a" )
17         << " control character\n"
18         << ( iscntrl( '$' ) ? "$ is a" : "$ is not a" )
19         << " control character\n";
20     cout << "\nAccording to ispunct:\n"
21         << ( ispunct( ';' ) ? "; is a" : "; is not a" )
22         << " punctuation character\n"
23         << ( ispunct( 'Y' ) ? "Y is a" : "Y is not a" )
24         << " punctuation character\n"
25         << ( ispunct( '#' ) ? "# is a" : "# is not a" )
26         << " punctuation character\n";
```

```

27     cout << "\nAccording to isprint:\n"
28         << ( isprint( '$' ) ? "$ is a" : "$ is not a" )
29         << " printing character\nAlert "
30         << ( isprint( '\a' ) ? "is a" : "is not a" )
31         << " printing character\n";
32     cout << "\nAccording to isgraph:\n"
33         << ( isgraph( 'Q' ) ? "Q is a" : "Q is not a" )
34         << " printing character other than a space\nSpace "
35         << ( isgraph( ' ' ) ? "is a" : "is not a" )
36         << " printing character other than a space" << endl;
37
38     return 0;
39 }

```

Fig. 16.19 Using `isspace`, `isctrl`, `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 isctrl:
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`, `isctrl`, `ispunct`, `isprint`, and `isgraph` (part 3 of 3).

Prototype	Description
<code>double atof(const char *nPtr)</code>	Converts the string <code>nPtr</code> to <code>double</code> .
<code>int atoi(const char *nPtr)</code>	Converts the string <code>nPtr</code> to <code>int</code> .
<code>long atol(const char *nPtr)</code>	Converts the string <code>nPtr</code> to long <code>int</code> .
<code>double strtod(const char *nPtr, char **endPtr)</code>	Converts the string <code>nPtr</code> to <code>double</code> .
<code>long strtol(const char *nPtr, char **endPtr, int base)</code>	Converts the string <code>nPtr</code> to long.
<code>unsigned long strtoul(const char *nPtr, char **endPtr, int base)</code>	Converts the string <code>nPtr</code> to <code>unsigned long</code> .

Fig. 16.20 Summary of the string conversion functions of the general utilities library.

```
1 // Fig. 16.21: fig16_21.cpp
2 // Using atof
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     double d = atof( "99.0" );
9
10    cout << "The string \"99.0\" converted to double is "
11         << d << "\nThe converted value divided by 2 is "
12         << d / 2.0 << endl;
13    return 0;
14 }
```

```
The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5
```

Fig. 16.21 Using `atof`.

```
1 // Fig. 16.22: fig16_22.cpp
2 // Using atoi
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     int i = atoi( "2593" );
9
10    cout << "The string \"2593\" converted to int is " << i
11         << "\nThe converted value minus 593 is " << i - 593
12         << endl;
13    return 0;
14 }
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Fig. 16.22 Using `atoi`.

```
1 // Fig. 16.23: fig16_23.cpp
2 // Using atol
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     long l = atol( "1000000" );
9
10    cout << "The string \"1000000\" converted to long is " << l
11         << "\nThe converted value divided by 2 is " << l / 2
12         << endl;
13    return 0;
14 }
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Fig. 16.23 Using `atol`.

```
1 // Fig. 16.24: fig16_24.cpp
2 // Using strtod
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     double d;
9     char *string = "51.2% are admitted", *stringPtr;
```

Fig. 16.24 Using `strtod` (part 1 of 2).

```
10
11     d = strtod( string, &stringPtr );
12     cout << "The string \"" << string
13         << "\" is converted to the\ndouble value " << d
14         << " and the string \"" << stringPtr << "\" << endl;
15     return 0;
16 }
```

```
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).

```
1 // Fig. 16.25: fig16_25.cpp
2 // Using strtol
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     long x;
9     char *string = "-1234567abc", *remainderPtr;
```

Fig. 16.25 Using `strtol` (part 1 of 2).

```
10
11     x = strtol( string, &remainderPtr, 0 );
12     cout << "The original string is \"" << string
13         << "\"\n\nThe converted value is " << x
14         << "\n\nThe remainder of the original string is \""
15         << remainderPtr
16         << "\"\n\nThe converted value plus 567 is "
17         << x + 567 << endl;
18     return 0;
19 }
```

```
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).

```
1 // Fig. 16.26: fig16_26.cpp
2 // Using strtoul
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     unsigned long x;
9     char *string = "1234567abc", *remainderPtr;
10
11     x = strtoul( string, &remainderPtr, 0 );
12     cout << "The original string is \" << string
13         << "\"\nThe converted value is " << x
14         << "\nThe remainder of the original string is \""
15         << remainderPtr
16         << "\"\nThe converted value minus 567 is "
17         << x - 567 << endl;
18     return 0;
19 }
```

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
<code>char *strchr(const char *s, int c)</code>	Locates the first occurrence of character <code>c</code> in string <code>s</code> . If <code>c</code> is found, a pointer to <code>c</code> in <code>s</code> is returned. Otherwise, a <code>NULL</code> pointer is returned.
<code>size_t strcspn(const char *s1, const char *s2)</code>	Determines and returns the length of the initial segment of string <code>s1</code> consisting of characters not contained in string <code>s2</code> .
<code>size_t strspn(const char *s1, const char *s2)</code>	Determines and returns the length of the initial segment of string <code>s1</code> consisting only of characters contained in string <code>s2</code> .
<code>char *strpbrk(const char *s1, const char *s2)</code>	

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.
<code>char *strrchr(const char *s, int c)</code>	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.
<code>char *strstr(const char *s1, const char *s2)</code>	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.

```

1 // Fig. 16.28: fig16_28.cpp
2 // Using strchr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string = "This is a test";
9     char character1 = 'a', character2 = 'z';
10
11     if ( strchr( string, character1 ) != NULL )
12         cout << '\'' << character1 << " was found in \""
13             << string << "\".\n";
14     else
15         cout << '\'' << character1 << " was not found in \""
16             << string << "\".\n";
17
18     if ( strchr( string, character2 ) != NULL )
19         cout << '\'' << character2 << " was found in \""
20             << string << "\".\n";
21     else
22         cout << '\'' << character2 << " was not found in \""
23             << string << "\"." << endl;
24     return 0;
25 }
```

```

'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Fig. 16.28 Using `strchr`.

```

1 // Fig. 16.29: fig16_29.cpp
2 // Using strcspn
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "The value is 3.14159";
9     char *string2 = "1234567890";
```

```

10
11     cout << "string1 = " << string1 << "\nstring2 = " << string2
12         << "\n\nThe length of the initial segment of string1"
13         << "\n\ncontaining no characters from string2 = "
14         << strcspn( string1, string2 ) << endl;
15     return 0;
16 }

```

```

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`.

```

1 // Fig. 16.30: fig16_30.cpp
2 // Using strpbrk
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "This is a test";
9     char *string2 = "beware";
10
11     cout << "Of the characters in \">

```

```

Of the characters in "beware"
'a' is the first character to appear in
"This is a test"

```

Fig. 16.30 Using `strpbrk`.

```

1 // Fig. 16.31: fig16_31.cpp
2 // Using strchr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "A zoo has many animals including zebras";
9     int c = 'z';
10
11     cout << "The remainder of string1 beginning with the\n"
12         << "last occurrence of character '" << (char) c
13         << "' is: \"" << strchr( string1, c ) << "'\n" << endl;
14     return 0;
15 }

```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Fig. 16.31 Using `strchr`.

```
1 // Fig. 16.32: fig16_32.cpp
2 // Using strspn
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "The value is 3.14159";
9     char *string2 = "aehilsTuv ";
10
11     cout << "string1 = " << string1
12         << "\nstring2 = " << string2
13         << "\n\nThe length of the initial segment of string1\n"
14         << "containing only characters from string2 = "
15         << strspn( string1, string2 ) << endl;
16     return 0;
17 }
```

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).

```
1 // Fig. 16.33: fig16_33.cpp
2 // Using strstr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *string1 = "abcdefabcdef";
9     char *string2 = "def";
10
11     cout << "string1 = " << string1 << "\nstring2 = " << string2
12         << "\n\nThe remainder of string1 beginning with the\n"
13         << "first occurrence of string2 is: "
14         << strstr( string1, string2 ) << endl;
15     return 0;
16 }
```



```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Fig. 16.33 Using `strstr`.

Prototype	Description
<code>void *memcpy(void *s1, const void *s2, size_t n)</code>	Copies <code>n</code> characters from the object pointed to by <code>s2</code> into the object pointed to by <code>s1</code> . A pointer to the resulting object is returned.
<code>void *memmove(void *s1, const void *s2, size_t n)</code>	Copies <code>n</code> characters from the object pointed to by <code>s2</code> into the object pointed to by <code>s1</code> . The copy is performed as if the characters are first copied from the object pointed to by <code>s2</code> into a temporary array, then from the temporary array into the object pointed to by <code>s1</code> . A pointer to the resulting object is returned.
<code>int memcmp(const void *s1, const void *s2, size_t n)</code>	Compares the first <code>n</code> characters of the objects pointed to by <code>s1</code> and <code>s2</code> . The function returns <code>0</code> , less than <code>0</code> , or greater than <code>0</code> if <code>s1</code> is equal to, less than, or greater than <code>s2</code> .
<code>void *memchr(const void *s, int c, size_t n)</code>	Locates the first occurrence of <code>c</code> (converted to <code>unsigned char</code>) in the first <code>n</code> characters of the object pointed to by <code>s</code> . If <code>c</code> is found, a pointer to <code>c</code> in the object is returned. Otherwise, <code>0</code> is returned.
<code>void *memset(void *s, int c, size_t n)</code>	Copies <code>c</code> (converted to <code>unsigned char</code>) into the first <code>n</code> characters of the object pointed to by <code>s</code> . A pointer to the result is returned.

Fig. 16.34 The memory functions of the string handling library.

```
1 // Fig. 16.35: fig16_35.cpp
2 // Using memcpy
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char s1[ 17 ], s2[] = "Copy this string";
9
10    memcpy( s1, s2, 17 );
11    cout << "After s2 is copied into s1 with memcpy,\n"
12         << "s1 contains \"" << s1 << "\"\n" << endl;
13    return 0;
14 }
```

After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"

Fig. 16.35 Using memcpy.

```

1 // Fig. 16.36: fig16_36.cpp
2 // Using memmove
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char x[] = "Home Sweet Home";
9
10    cout << "The string in array x before memmove is: " << x;
11    cout << "\nThe string in array x after memmove is: "
12         << (char *) memmove( x, &x[ 5 ], 10 ) << endl;
13    return 0;
14 }
```

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.

```

1 // Fig. 16.37: fig16_37.cpp
2 // Using memcmp
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <string.h>
6
7 int main()
8 {
9     char s1[] = "ABCDEFGH", s2[] = "ABCDXYZ";
10
11    cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
12         << "\nmemcmp(s1, s2, 4) = " << setw( 3 )
13         << memcmp( s1, s2, 4 ) << "\nmemcmp(s1, s2, 7) = "
14         << setw( 3 ) << memcmp( s1, s2, 7 )
15         << "\nmemcmp(s2, s1, 7) = " << setw( 3 )
16         << memcmp( s2, s1, 7 ) << endl;
17    return 0;
18 }
```

s1 = ABCDEFGH
s2 = ABCDXYZ

memcmp(s1, s2, 4) = 0
memcmp(s1, s2, 7) = -19
memcmp(s2, s1, 7) = 19

Fig. 16.37 Using memcmp.

```

1 // Fig. 16.38: fig16_38.cpp
2 // Using memchr
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char *s = "This is a string";

```

Fig. 16.38 Using `memchr` (part 1 of 2).

```

9
10     cout << "The remainder of s after character 'r' "
11         << "is found is \"" << (char *) memchr( s, 'r', 16 )
12         << "\" " << endl;
13     return 0;
14 }

```

```
The remainder of s after character 'r' is found is "ring"
```

Fig. 16.38 Using `memchr` (part 2 of 2).

```

1 // Fig. 16.39: fig16_39.cpp
2 // Using memset
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     char string1[ 15 ] = "BBBBBBBBBBBBBB";
9
10     cout << "string1 = " << string1 << endl;
11     cout << "string1 after memset = "
12         << (char *) memset( string1, 'b', 7 ) << endl;
13     return 0;
14 }

```

```
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.

```
1 // Fig. 16.41: fig16_41.cpp
2 // Using strerror
3 #include <iostream.h>
4 #include <string.h>
5
6 int main()
7 {
8     cout << strerror( 2 ) << endl;
9     return 0;
10 }
```



No such file or directory

Fig. 16.41 Using `strerror`.

Illustrations List (Main Page)**Fig. 17.1** The predefined symbolic constants.

Symbolic constant	Description
<code>__LINE__</code>	The line number of the current source code line (an integer constant).
<code>__FILE__</code>	The presumed name of the source file (a string).
<code>__DATE__</code>	The date the source file is compiled (a string of the form " Mmm dd yyyy " such as " Jan 19 1994 ").
<code>__TIME__</code>	The time the source file is compiled (a string literal of the form " hh:mm:ss ").
<code>__STDC__</code>	The integer constant 1. This is intended to indicate that the implementation is ANSI compliant.

Fig. 17.1 The predefined symbolic constants.

Illustrations List [\(Main Page\)](#)

- Fig. 18.1** The type and the macros defined in header **stdarg.h**.
- Fig. 18.2** Using variable-length argument lists.
- Fig. 18.3** Using command-line arguments.
- Fig. 18.4** Using functions **exit** and **atexit**.
- Fig. 18.5** The signals defined in header **signal.h**.
- Fig. 18.6** Using signal handling.
- Fig. 18.7** Using **goto**.
- Fig. 18.8** Printing the value of a union in both member data types.
- Fig. 18.9** Using an anonymous **union**.

Identifier	Description
<code>va_list</code>	A type suitable for holding information needed by macros <code>va_start</code> , <code>va_arg</code> , and <code>va_end</code> . To access the arguments in a variable-length argument list, an object of type <code>va_list</code> must be declared.
<code>va_start</code>	A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with <code>va_list</code> for use by the <code>va_arg</code> and <code>va_end</code> macros.
<code>va_arg</code>	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 <code>va_arg</code> modifies the object declared with <code>va_list</code> so that the object points to the next argument in the list.
<code>va_end</code>	A macro that facilitates a normal return from a function whose variable-length argument list was referred to by the <code>va_start</code> macro.

Fig. 18.1 The type and the macros defined in header `stdarg.h`.

```

1 // Fig. 18.2: fig18_02.cpp
2 // Using variable-length argument lists
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <stdarg.h>
6
7 double average( int, ... );
8
9 int main()
10 {
11     double w = 37.5, x = 22.5, y = 1.7, z = 10.2;
12
13     cout << setiosflags( ios::fixed | ios::showpoint )
14          << setprecision( 1 ) << "w = " << w << "\nx = " << x
15          << "\ny = " << y << "\nz = " << z << endl;
16     cout << setprecision( 3 ) << "\nThe average of w and x is "
17          << average( 2, w, x )
18          << "\nThe average of w, x, and y is "
19          << average( 3, w, x, y )
20          << "\nThe average of w, x, y, and z is "
21          << average( 4, w, x, y, z ) << endl;
22     return 0;
23 }
24
25 double average( int i, ... )
26 {
27     double total = 0;

```

Fig. 18.2 Using variable-length argument lists (part 1 of 2).

```

28     va_list ap;
29
30     va_start( ap, i );
31

```

```

32     for ( int j = 1; j <= i; j++ )
33         total += va_arg( ap, double );
34
35     va_end( ap );
36
37     return total / i;
38 }

```

```

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).

```

1 // Fig. 18.3: fig18_03.cpp
2 // Using command-line arguments
3 #include <iostream.h>
4 #include <fstream.h>
5
6 int main( int argc, char *argv[] )
7 {
8     if ( argc != 3 )
9         cout << "Usage: copy infile outfile" << endl;
10    else {
11        ifstream inFile( argv[ 1 ], ios::in );
12        if ( !inFile )
13            cout << argv[ 1 ] << " could not be opened" << endl;
14
15        ofstream outFile( argv[ 2 ], ios::out );
16        if ( !outFile )
17            cout << argv[ 2 ] << " could not be opened" << endl;
18
19        while ( !inFile.eof() )
20            outFile.put( static_cast< char >( inFile.get() ) );
21    }
22
23    return 0;
24 }

```

Fig. 18.3 Using command-line arguments.

```

1 // Fig. 18.4: fig18_04.cpp
2 // Using the exit and atexit functions
3 #include <iostream.h>
4 #include <stdlib.h>
5
6 void print( void );
7
8 int main()
9 {
10    atexit( print ); // register function print
11    cout << "Enter 1 to terminate program with function exit"
12         << "\nEnter 2 to terminate program normally\n";

```



```

13
14     int answer;
15     cin >> answer;
16
17     if ( answer == 1 ) {
18         cout << "\nTerminating program with function exit\n";
19         exit( EXIT_SUCCESS );
20     }
21
22     cout << "\nTerminating program by reaching the of main"
23         << endl;
24
25     return 0;
26 }
27
28 void print( void )
29 {
30     cout << "Executing function print at program termination\n"
31         << "Program terminated" << endl;
32 }

```

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`.

```
1 // Fig. 18.6: fig18_06.cpp
2 // Using signal handling
3 #include <iostream.h>
4 #include <iomanip.h>
5 #include <signal.h>
6 #include <stdlib.h>
7 #include <time.h>
8
9 void signal_handler( int );
10
11 int main()
12 {
13     signal( SIGINT, signal_handler );
14     srand( time( 0 ) );
15
16     for ( int i = 1; i < 101; i++ ) {
17         int x = 1 + rand() % 50;
18
19         if ( x == 25 )
20             raise( SIGINT );
21
22         cout << setw( 4 ) << i;
23
24         if ( i % 10 == 0 )
25             cout << endl;
26     }
27
28     return 0;
29 }
30
31 void signal_handler( int signalValue )
32 {
33     cout << "\nInterrupt signal (" << signalValue
34         << ") received.\n"
35         << "Do you wish to continue (1 = yes or 2 = no)? ";
36
37     int response;
38     cin >> response;
39
40     while ( response != 1 && response != 2 ) {
41         cout << "(1 = yes or 2 = no)? ";
42         cin >> response;
43     }
44
45     if ( response == 1 )
46         signal( SIGINT, signal_handler );
47     else
48         exit( EXIT_SUCCESS );
49 }
```

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).

```

1 // Fig. 18.7: fig18_07.cpp
2 // Using goto
3 #include <iostream.h>
4
5 int main()
6 {
7     int count = 1;
8
9     start:                // label
10    if ( count > 10 )
11        goto end;
12
13    cout << count << " ";
14    ++count;
15    goto start;
16
17    end:                  // label
18    cout << endl;
19
20    return 0;
21 }

```

```

1 2 3 4 5 6 7 8 9 10

```

Fig. 18.7 Using goto.

```

1 // Fig. 18.8: fig18_08.cpp
2 // An example of a union
3 #include <iostream.h>
4
5 union Number {
6     int x;
7     float y;
8 };

```

Fig. 18.8 Printing the value of a **union** in both member data types (part 1 of 2).

```

9
10 int main()
11 {
12     Number value;
13
14     value.x = 100;
15     cout << "Put a value in the integer member\n"
16           << "and print both members.\nint:  "
17           << value.x << "\nfloat: " << value.y << "\n\n";
18
19     value.y = 100.0;
20     cout << "Put a value in the floating member\n"
21           << "and print both members.\nint:  "
22           << value.x << "\nfloat: " << value.y << endl;
23     return 0;
24 }

```

```

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).

```

1 // Fig. 18.9: fig18_09.cpp
2 // Using an anonymous union
3 #include <iostream.h>
4
5 int main()
6 {
7     // Declare an anonymous union.

```

Fig. 18.9 Using an anonymous **union** (part 1 of 2).

```

8     // Note that members b, d, and f share the same space.
9     union {
10         int b;
11         double d;
12         char *f;
13     };
14
15     // Declare conventional local variables
16     int a = 1;
17     double c = 3.3;
18     char *e = "Anonymous";
19
20     // Assign a value to each union member
21     // successively and print each.
22     cout << a << ' ';
23     b = 2;
24     cout << b << endl;
25
26     cout << c << ' ';
27     d = 4.4;
28     cout << d << endl;

```

```
29
30     cout << e << ' ';
31     f = "union";
32     cout << f << endl;
33
34     return 0;
35 }
```

```
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** Demonstrating `string` assignment and concatenation.
- Fig. 19.2** Comparing `strings`.
- Fig. 19.3** Using function `substr` to extract a substring from a `string`.
- Fig. 19.4** Using function `swap` to swap two `strings`.
- Fig. 19.5** Printing `string` characteristics.
- Fig. 19.6** Program that demonstrates the `string find` functions.
- Fig. 19.7** Demonstrating functions `erase` and `replace`.
- Fig. 19.8** Demonstrating the `string insert` functions.
- Fig. 19.9** Converting `strings` to C-style strings and character arrays.
- Fig. 19.10** Using an iterator to output a `string`.
- Fig. 19.11** Using a dynamically allocated `ostream` object.
- Fig. 19.12** Demonstrating input from an `istream` object.

```
1 // Fig. 19.1: fig19_01.cpp
2 // Demonstrating string assignment and concatenation
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "cat" ), s2, s3;
10
11     s2 = s1;          // assign s1 to s2 with =
12     s3.assign( s1 ); // assign s1 to s3 with assign()
13     cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: "
14         << s3 << "\n\n";
15
16     // modify s2 and s3
17     s2[ 0 ] = s3[ 2 ] = 'r';
18
19     cout << "After modification of s2 and s3:\n"
20         << "s1: " << s1 << "\ns2: " << s2 << "\ns3: ";
21
22     // demonstrating member function at()
23     int len = s3.length();
24     for ( int x = 0; x < len; ++x )
25         cout << s3.at( x );
26
27     // concatenation
28     string s4( s1 + "apult" ), s5; // declare s4 and s5
29
30     // overloaded +=
31     s3 += "pet";          // create "carpet"
32     s1.append( "acomb" ); // create "catacomb"
33
34     // append subscript locations 4 thru the end of s1 to
35     // create the string "comb" (s5 was initially empty)
36     s5.append( s1, 4, s1.size() );
37
38     cout << "\n\nAfter concatenation:\n" << "s1: " << s1
39         << "\ns2: " << s2 << "\ns3: " << s3 << "\ns4: " << s4
40         << "\ns5: " << s5 << endl;
41
42     return 0;
43 }
```

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).

```

1 // Fig. 19.2: fig19_02.cpp
2 // Demonstrating string comparison capabilities
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "Testing the comparison functions." ),
10            s2("Hello" ), s3( "stinger" ), z1( s2 );
11
12     cout << "s1: " << s1 << "\ns2: " << s2
13          << "\ns3: " << s3 << "\nz1: " << z1 << "\n\n";

```

Fig. 19.2 Comparing strings (part 1 of 3).

```

14
15 // comparing s1 and z1
16 if ( s1 == z1 )
17     cout << "s1 == z1\n";
18 else { // s1 != z1
19     if ( s1 > z1 )
20         cout << "s1 > z1\n";
21     else // s1 < z1
22         cout << "s1 < z1\n";
23 }
24
25 // comparing s1 and s2
26 int f = s1.compare( s2 );
27
28 if ( f == 0 )
29     cout << "s1.compare( s2 ) == 0\n";
30 else if ( f > 0 )
31     cout << "s1.compare( s2 ) > 0\n";
32 else // f < 0
33     cout << "s1.compare( s2 ) < 0\n";
34
35 // comparing s1 (elements 2 - 5) and s3 (elements 0 - 5)
36 f = s1.compare( 2, 5, s3, 0, 5 );
37
38 if ( f == 0 )

```



```

39     cout << "s1.compare( 2, 5, s3, 0, 5 ) == 0\n";
40     else if ( f > 0 )
41         cout << "s1.compare( 2, 5, s3, 0, 5 ) > 0\n";
42     else // f < 0
43         cout << "s1.compare( 2, 5, s3, 0, 5 ) < 0\n";
44
45     // comparing s2 and z1
46     f = z1.compare( 0, s2.size(), s2 );
47
48     if ( f == 0 )
49         cout << "z1.compare( 0, s2.size(), s2 ) == 0" << endl;
50     else if ( f > 0 )
51         cout << "z1.compare( 0, s2.size(), s2 ) > 0" << endl;
52     else // f < 0
53         cout << "z1.compare( 0, s2.size(), s2 ) < 0" << endl;
54
55     return 0;
56 }

```

Fig. 19.2 Comparing `strings` (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 `strings` (part 3 of 3).

```

1 // Fig. 19.3: fig19_03.cpp
2 // Demonstrating function substr
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s( "The airplane flew away." );
10
11     // retrieve the substring "plane" which
12     // begins at subscript 7 and consists of 5 elements
13     cout << s.substr( 7, 5 ) << endl;
14
15     return 0;
16 }

```

```
plane
```

Fig. 19.3 Using function `substr` to extract a substring from a `string`.

```

1 // Fig. 19.4: fig19_04.cpp
2 // Using the swap function to swap two strings
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string first( "one" ), second( "two" );
10
11     cout << "Before swap:\n first: " << first
12         << "\nsecond: " << second;
13     first.swap( second );
14     cout << "\n\nAfter swap:\n first: " << first
15         << "\nsecond: " << second << endl;
16
17     return 0;
18 }

```

```

Before swap:
 first: one
second: two

After swap:
 first: two
second: one

```

Fig. 19.4 Using function `swap` to swap two `strings`.

```

1 // Fig. 19.5: fig19_05.cpp
2 // Demonstrating functions related to size and capacity
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 void printStats( const string & );

```

Fig. 19.5 Printing `string` characteristics (part 1 of 3).

```

8
9 int main()
10 {
11     string s;
12
13     cout << "Stats before input:\n";
14     printStats( s );
15
16     cout << "\n\nEnter a string: ";
17     cin >> s; // delimited by whitespace
18     cout << "The string entered was: " << s;
19
20     cout << "\nStats after input:\n";
21     printStats( s );
22
23     s.resize( s.length() + 10 );
24     cout << "\n\nStats after resizing by (length + 10):\n";
25     printStats( s );
26
27     cout << endl;
28     return 0;

```

```

29 }
30
31 void printStats( const string &str )
32 {
33     cout << "capacity: " << str.capacity()
34         << "\nmax size: " << str.max_size()
35         << "\nsize: " << str.size()
36         << "\nlength: " << str.length()
37         << "\nempty: " << ( str.empty() ? "true": "false" );
38 }

```

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).

```

1 // Fig. 19.6: fig19_06.cpp
2 // Demonstrating the string find functions
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     // compiler concatenates all parts into one string literal
10    string s( "The values in any left subtree"
11            "\nare less than the value in the"
12            "\nparent node and the values in"
13            "\nany right subtree are greater"
14            "\nthan the value in the parent node" );

```

Fig. 19.6 Program that demonstrates the `string find` functions (part 1 of 2).

```

15
16    // find "subtree" at locations 23 and 102
17    cout << "Original string:\n" << s

```

```

18     << "\n\n(find) \"subtree\" was found at: "
19     << s.find( "subtree" )
20     << "\n(rfind) \"subtree\" was found at: "
21     << s.rfind( "subtree" );
22
23 // find 'p' in parent at locations 62 and 144
24 cout << "\n(find_first_of) character from \"qpxz\" at: "
25     << s.find_first_of( "qpxz" )
26     << "\n(find_last_of) character from \"qpxz\" at: "
27     << s.find_last_of( "qpxz" );
28
29 // find 'b' at location 25
30 cout << "\n(find_first_not_of) first character not\n"
31     << "    contained in \"heTv lusinodrpayft\": "
32     << s.find_first_not_of( "heTv lusinodrpayft" );
33
34 // find '\n' at location 121
35 cout << "\n(find_last_not_of) first character not\n"
36     << "    contained in \"heTv lusinodrpayft\": "
37     << s.find_last_not_of( "heTv lusinodrpayft" ) << endl;
38
39 return 0;
40 }

```

```

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).

```

1 // Fig. 19.7: fig19_07.cpp
2 // Demonstrating functions erase and replace
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     // compiler concatenates all parts into one string
10    string s( "The values in any left subtree"
11            "\nare less than the value in the"
12            "\nparent node and the values in"
13            "\nany right subtree are greater"
14            "\nthan the value in the parent node" );
15
16    // remove all characters from location 62
17    // through the end of s
18    s.erase( 62 );
19
20    // output the new string
21    cout << "Original string after erase:\n" << s

```

```

22     << "\n\nAfter first replacement:\n";
23
24 // replace all spaces with a period
25 int x = s.find( " " );
26 while ( x < string::npos ) {
27     s.replace( x, 1, "." );
28     x = s.find( " ", x + 1 );
29 }
30
31 cout << s << "\n\nAfter second replacement:\n";
32
33 // replace all periods with two semicolons
34 // NOTE: this will overwrite characters
35 x = s.find( "." );
36 while ( x < string::npos ) {
37     s.replace( x, 2, "xxxxx;yyy", 5, 2 );
38     x = s.find( ".", x + 1 );
39 }
40
41 cout << s << endl;
42 return 0;
43 }

```

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).

```

1 // Fig. 19.8: fig19_08.cpp
2 // Demonstrating the string insert functions.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "beginning end" ),
10         s2( "middle " ), s3( "12345678" ), s4( "xx" );
11
12     cout << "Initial strings:\ns1: " << s1
13         << "\ns2: " << s2 << "\ns3: " << s3
14         << "\ns4: " << s4 << "\n\n";
15
16     // insert "middle" at location 10
17     s1.insert( 10, s2 );
18
19     // insert "xx" at location 3 in s3
20     s3.insert( 3, s4, 0, string::npos );

```

```

21
22     cout << "Strings after insert:\ns1: " << s1
23         << "\ns2: " << s2 << "\ns3: " << s3
24         << "\ns4: " << s4 << endl;
25
26     return 0;
27 }

```

```

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.

```

1 // Fig. 19.9: fig19_09.cpp
2 // Converting to C-style strings.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s( "STRINGS" );
10    const char *ptr1 = 0;
11    int len = s.length();
12    char *ptr2 = new char[ len + 1 ]; // including null
13
14    // Assign to pointer ptr1 the const char * returned by
15    // function data(). NOTE: this is a potentially dangerous
16    // assignment. If the string is modified, the pointer
17    // ptr1 can become invalid.
18    ptr1 = s.data();
19
20    // copy characters out of string into allocated memory
21    s.copy( ptr2, len, 0 );
22    ptr2[ len ] = 0; // add null terminator
23
24    // output
25    cout << "string s is " << s
26         << "\ns converted to a C-Style string is "
27         << s.c_str() << "\nptr1 is ";
28
29    for ( int k = 0; k < len; ++k )
30        cout << *( ptr1 + k ); // use pointer arithmetic
31
32    cout << "\nptr2 is " << ptr2 << endl;
33    delete [] ptr2;
34    return 0;
35 }

```

Fig. 19.9 Converting `strings` 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).

```
1 // Fig. 19.10: fig19_10.cpp
2 // Using an iterator to output a string.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s( "Testing iterators" );
10    string::const_iterator il = s.begin();
11
12    cout << "s = " << s
13         << "\n(Using iterator il) s is: ";
14
15    while ( il != s.end() ) {
16        cout << *il;    // dereference iterator to get char
17        ++il;          // advance iterator to next char
18    }
19
20    cout << endl;
21    return 0;
22 }
```

```
s = Testing iterators
(Using iterator il) s is: Testing iterators
```

Fig. 19.10 Using an iterator to output a `string`.

```
1 // Fig. 19.11: fig19_11.cpp
2 // Using a dynamically allocated ostream object.
3 #include <iostream>
4 #include <string>
5 #include <sstream>
6 using namespace std;
7
8 main()
9 {
10    ostream outputString;
11    string s1( "Output of several data types " ),
12           s2( "to an ostream object:" ),
13           s3( "\n         double: " ),
14           s4( "\n         int: " ),
15           s5( "\naddress of int: " );
16    double d = 123.4567;
17    int i = 22;
18
19    outputString << s1 << s2 << s3 << d << s4 << i << s5 << &i;
20    cout << "outputString contains:\n" << outputString.str();
21 }
```

```

22     outputStream << "\nmore characters added";
23     cout << "\n\nafter additional stream insertions,\n"
24         << "outputString contains:\n" << outputStream.str()
25         << endl;
26
27     return 0;
28 }

```

```

outputString contains:
Output of several data types to an ostream object:
    double: 123.457
        int: 22
address of int: 0068FD0C

after additional stream insertions,
outputString contains:
Output of several data types to an ostream object:
    double: 123.457
        int: 22
address of int: 0068FD0C
more characters added

```

Fig. 19.11 Using a dynamically allocated `ostream` object.

```

1 // Fig. 19.12: fig19_12.cpp
2 // Demonstrating input from an istream object.
3 #include <iostream>
4 #include <string>
5 #include <sstream>
6 using namespace std;
7

```

Fig. 19.12 Demonstrating input from an `istream` object (part 1 of 2).

```

8     main()
9     {
10        string input( "Input test 123 4.7 A" );
11        istream inputString( input );
12        string string1, string2;
13        int i;
14        double d;
15        char c;
16
17        inputString >> string1 >> string2 >> i >> d >> c;
18
19        cout << "The following inputs were extracted\n"
20             << "from the istream object:"
21             << "\nstring: " << string1
22             << "\nstring: " << string2
23             << "\n int: " << i
24             << "\ndouble: " << d
25             << "\n char: " << c;
26
27        // attempt to read from empty stream
28        long l;
29
30        if ( inputString >> l )
31            cout << "\n\nlong value is: " << l << endl;
32        else

```



```
33     cout << "\n\ninputString is empty" << endl;  
34  
35     return 0;  
36 }
```

```
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)

- Fig. 20.1 Standard Library container classes.
- Fig. 20.2 Common functions for all STL containers.
- Fig. 20.3 Standard Library container header files.
- Fig. 20.4 Common **typedefs** found in first-class containers.
- Fig. 20.5 Demonstrating input and output stream iterators .
- Fig. 20.6 Iterator categories.
- Fig. 20.7 Iterator category hierarchy.
- Fig. 20.8 Iterator types supported by each Standard Library container.
- Fig. 20.9 Predefined iterator **typedefs**.
- Fig. 20.10 Iterator operations for each type of iterator .
- Fig. 20.11 Mutating-sequence algorithms.
- Fig. 20.12 Non-mutating sequence algorithms.
- Fig. 20.13 Numerical algorithms from header file **<numeric>**.
- Fig. 20.14 Demonstrating Standard Library **vector** class template.
- Fig. 20.15 Demonstrating Standard Library **vector** class template element-manipulation functions.
- Fig. 20.16 STL exception types.
- Fig. 20.17 Demonstrating Standard Library **list** class template.
- Fig. 20.18 Demonstrating Standard Library **deque** class template.
- Fig. 20.19 Demonstrating Standard Library **multiset** class template.
- Fig. 20.20 Demonstrating Standard Library **set** class template.
- Fig. 20.21 Demonstrating Standard Library **multimap** class template.
- Fig. 20.22 Demonstrating Standard Library **map** class template.
- Fig. 20.23 Demonstrating Standard Library **stack** adapter class.
- Fig. 20.24 Demonstrating Standard Library **queue** adapter class templates.
- Fig. 20.25 Demonstrating Standard Library **priority_queue** adapter class.
- Fig. 20.26 Demonstrating Standard Library functions **fill**, **fill_n**, **generate** and **generate_n**.
- Fig. 20.27 Demonstrating Standard Library functions **equal**, **mismatch** and **lexicographical_compare**.
- Fig. 20.28 Demonstrating Standard Library functions **remove**, **remove_if**, **remove_copy** and **remove_copy_if**.
- Fig. 20.29 Demonstrating Standard Library functions **replace**, **replace_if**, **replace_copy** and **replace_copy_if**.
- Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library .
- Fig. 20.31 Basic searching and sorting algorithms of the Standard Library.
- Fig. 20.32 Demonstrating **swap**, **iter_swap** and **swap_ranges**.
- Fig. 20.33 Demonstrating **copy_backward**, **merge**, **unique** and **reverse**.
- Fig. 20.34 Demonstrating **inplace_merge**, **unique_copy** and **reverse_copy**.
- Fig. 20.35 Demonstrating **set** operations of the Standard Library .
- Fig. 20.36 Demonstrating **lower_bound**, **upper_bound** and **equal_range**.
- Fig. 20.37 Using Standard Library functions to perform a heapsort.
- Fig. 20.38 Demonstrating algorithms **min** and **max**.
- Fig. 20.39 Algorithms not covered in this chapter.
- Fig. 20.40 Demonstrating class **bitset** and the Sieve of Eratosthenes.
- Fig. 20.41 Function objects in the Standard Library .
- Fig. 20.42 Demonstrating a binary function object .

Standard Library container class	Description
<i>Sequence Containers</i>	
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
<i>Associative Containers</i>	
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
<i>Container Adapters</i>	
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 if 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
<code>operator>=</code>	Returns true if the first container is greater than or equal to the second container; otherwise, returns false .
<code>operator==</code>	Returns true if the first container is equal to the second container; otherwise, returns false .
<code>operator!=</code>	Returns true if the first container is not equal to the second container; otherwise, returns false .
<code>swap</code>	Swaps the elements of two containers.
<i>Functions that are only found in first-class containers</i>	
<code>begin</code>	The two versions of this function return either an iterator or a const_iterator that refers to the first element of the container.
<code>end</code>	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.
<code>rbegin</code>	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.
<code>rend</code>	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.
<code>erase</code>	Erases one or more elements from the container.
<code>clear</code>	Erases all elements from the container.

Fig. 20.2 Common functions for all STL containers.

Standard Library container header files	
<code><vector></code>	
<code><list></code>	
<code><deque></code>	
<code><queue></code>	contains both queue and priority_queue
<code><stack></code>	
<code><map></code>	contains both map and multimap
<code><set></code>	contains both set and multiset
<code><bitset></code>	

Fig. 20.3 Standard Library container header files.

typedef	Description
<code>value_type</code>	The type of element stored in the container.
<code>reference</code>	A reference to the type of element stored in the container.
<code>const_reference</code>	A constant reference to the type of element stored in the container. Such a reference can only be used for <i>reading</i> elements in the container and for performing const operations.
<code>pointer</code>	A pointer to the type of element stored in the container.
<code>iterator</code>	An iterator that points to the type of element stored in the container.
<code>const_iterator</code>	A constant iterator that points to the type of element stored in the container and can only be used to <i>read</i> elements.
<code>reverse_iterator</code>	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.
<code>const_reverse_iterator</code>	A constant reverse iterator to the type of element stored in the container and can only be used to <i>read</i> elements. This type of iterator is for iterating through a container in reverse.
<code>difference_type</code>	The type of the result of subtracting two iterators that refer to the same container (operator- is not defined for iterators of lists and associative containers).
<code>size_type</code>	The type used to count items in a container and index through a sequence container (cannot index through a list).

Fig. 20.4 Common typedefs found in first-class containers.

```

1 // Fig. 20.5: fig20_05.cpp
2 // Demonstrating input and output with iterators.
3 #include <iostream>
4 #include <iterator>
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "Enter two integers: ";
11

```

Fig. 20.5 Demonstrating input and output stream iterators (part 1 of 2).

```

12     istream_iterator< int > inputInt( cin );
13     int number1, number2;
14
15     number1 = *inputInt; // read first int from standard input
16     ++inputInt;         // move iterator to next input value
17     number2 = *inputInt; // read next int from standard input
18
19     cout << "The sum is: ";
20
21     ostream_iterator< int > outputInt( cout );
22
23     *outputInt = number1 + number2; // output result to cout

```

```

24     cout << endl;
25     return 0;
26 }

```

```

Enter two integers: 12 25
The sum is: 37

```

Fig. 20.5 Demonstrating input and output stream iterators (part 2 of 2).

Category	Description
<i>input</i>	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.
<i>output</i>	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.
<i>forward</i>	Combines the capabilities of input and output iterators and retain their position in the container (as state information).
<i>bidirectional</i>	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.
<i>random access</i>	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.

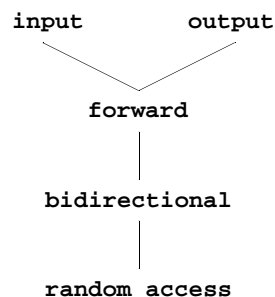


Fig. 20.7 Iterator category hierarchy.

Container	Type of iterator supported
<i>Sequence containers</i>	
vector	random access
deque	random access
list	bidirectional
<i>Associative containers</i>	
set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional
<i>Container adapters</i>	
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 typedefs 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 typedefs.

Iterator operation	Description
<i>All iterators</i>	
++p	preincrement an iterator
p++	postincrement an iterator
<i>Input iterators</i>	
*p	dereference an iterator for use as an <i>rvalue</i>
p = p1	assign one iterator to another
p == p1	compare iterators for equality
p != p1	compare iterators for inequality
<i>Output iterators</i>	
*p	dereference an iterator (for use as an <i>lvalue</i>)

Fig. 20.10 Iterator operations for each type of iterator.

Iterator operation	Description
<code>p = p1</code>	assign one iterator to another
<i>Forward iterators</i>	Forward iterators provide all the functionality of both input iterators and output iterators.
<i>Bidirectional iterators</i>	
<code>--p</code>	predecrement an iterator
<code>p--</code>	postdecrement an iterator
<i>Random-access iterators</i>	
<code>p += i</code>	Increment the iterator <code>p</code> by <code>i</code> positions.
<code>p -= i</code>	Decrement the iterator <code>p</code> by <code>i</code> positions.
<code>p + i</code>	Results in an iterator positioned at <code>p</code> incremented by <code>i</code> positions.
<code>p - i</code>	Results in an iterator positioned at <code>p</code> decremented by <code>i</code> positions.
<code>p[i]</code>	Return a reference to the element offset from <code>p</code> by <code>i</code> positions
<code>p < p1</code>	Return true if iterator <code>p</code> is less than iterator <code>p1</code> (i.e., iterator <code>p</code> is before iterator <code>p1</code> in the container); otherwise, return false .
<code>p <= p1</code>	Return true if iterator <code>p</code> is less than or equal to iterator <code>p1</code> (i.e., iterator <code>p</code> is before iterator <code>p1</code> or at the same location as iterator <code>p1</code> in the container); otherwise, return false .
<code>p > p1</code>	Return true if iterator <code>p</code> is greater than iterator <code>p1</code> (i.e., iterator <code>p</code> is after iterator <code>p1</code> in the container); otherwise, return false .
<code>p >= p1</code>	Return true if iterator <code>p</code> is greater than or equal to iterator <code>p1</code> (i.e., iterator <code>p</code> is after iterator <code>p1</code> or at the same location as iterator <code>p1</code> in the container); otherwise, return false .

Fig. 20.10 Iterator operations for each type of iterator.

Mutating-sequence algorithms		
<code>copy()</code>	<code>remove()</code>	<code>reverse_copy()</code>
<code>copy_backward()</code>	<code>remove_copy()</code>	<code>rotate()</code>
<code>fill()</code>	<code>remove_copy_if()</code>	<code>rotate_copy()</code>
<code>fill_n()</code>	<code>remove_if()</code>	<code>stable_partition()</code>
<code>generate()</code>	<code>replace()</code>	<code>swap()</code>
<code>generate_n()</code>	<code>replace_copy()</code>	<code>swap_ranges()</code>
<code>iter_swap()</code>	<code>replace_copy_if()</code>	<code>transform()</code>
<code>partition()</code>	<code>replace_if()</code>	<code>unique()</code>
<code>random_shuffle()</code>	<code>reverse()</code>	<code>unique_copy()</code>

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>.

```
1 // Fig. 20.14: fig20_14.cpp
2 // Testing Standard Library vector class template
3 #include <iostream>
4 #include <vector>
5
6 using namespace std;
7
8 template < class T >
9 void printVector( const vector< T > &vec );
10
11 int main()
12 {
13     const int SIZE = 6;
14     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15     vector< int > v;
16
17     cout << "The initial size of v is: " << v.size()
18         << "\nThe initial capacity of v is: " << v.capacity();
19     v.push_back( 2 ); // method push_back() is in
20     v.push_back( 3 ); // every sequence container
21     v.push_back( 4 );
22     cout << "\nThe size of v is: " << v.size()
23         << "\nThe capacity of v is: " << v.capacity();
24     cout << "\n\nContents of array a using pointer notation: ";
25
26     for ( int *ptr = a; ptr != a + SIZE; ++ptr )
27         cout << *ptr << ' ';
28
29     cout << "\n\nContents of vector v using iterator notation: ";
30     printVector( v );
31
32     cout << "\n\nReversed contents of vector v: ";
33
34     vector< int >::reverse_iterator p2;
35
36     for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
37         cout << *p2 << ' ';
```

```

38     cout << endl;
39     return 0;
40 }

```

Fig. 20.14 Demonstrating Standard Library **vector** class template (part 1 of 2).

```

41
42 template < class T >
43 void printVector( const vector< T > &vec )
44 {
45     vector< T >::const_iterator p1;
46
47     for ( p1 = vec.begin(); p1 != vec.end(); ++p1 )
48         cout << *p1 << ' ';
49 }

```

```

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).

```

1 // Fig. 20.15: fig20_15.cpp
2 // Testing Standard Library vector class template
3 // element-manipulation functions
4 #include <iostream>
5 #include <vector>
6 #include <algorithm>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 6;
13     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
14     vector< int > v( a, a + SIZE );
15     ostream_iterator< int > output( cout, " " );
16     cout << "Vector v contains: ";
17     copy( v.begin(), v.end(), output );
18
19     cout << "\nFirst element of v: " << v.front()
20         << "\nLast element of v: " << v.back();
21
22     v[ 0 ] = 7;           // set first element to 7
23     v.at( 2 ) = 10;     // set element at position 2 to 10
24     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).

```

25     cout << "\nContents of vector v after changes: ";
26     copy( v.begin(), v.end(), output );

```

```

27
28     try {
29         v.at( 100 ) = 777;    // access element out of range
30     }
31     catch ( out_of_range e ) {
32         cout << "\nException: " << e.what();
33     }
34
35     v.erase( v.begin() );
36     cout << "\nContents of vector v after erase: ";
37     copy( v.begin(), v.end(), output );
38     v.erase( v.begin(), v.end() );
39     cout << "\nAfter erase, vector v "
40         << ( v.empty() ? "is" : "is not" ) << " empty";
41
42     v.insert( v.begin(), a, a + SIZE );
43     cout << "\nContents of vector v before clear: ";
44     copy( v.begin(), v.end(), output );
45     v.clear(); // clear calls erase to empty a collection
46     cout << "\nAfter clear, vector v "
47         << ( v.empty() ? "is" : "is not" ) << " empty";
48
49     cout << endl;
50     return 0;
51 }

```

```

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
<code>out_of_range</code>	Indicates when subscript is out of range—e.g., when an invalid subscript is specified to vector member function <code>at</code> .
<code>invalid_argument</code>	Indicates an invalid argument was passed to a function.
<code>length_error</code>	Indicates an attempt to create too long a container, string , etc.
<code>bad_alloc</code>	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.

```

1 // Fig. 20.17: fig20_17.cpp
2 // Testing Standard Library class list
3 #include <iostream>
4 #include <list>
5 #include <algorithm>
6
7 using namespace std;
8
9 template < class T >
10 void printList( const list< T > &listRef );
11
12 int main()
13 {
14     const int SIZE = 4;
15     int a[ SIZE ] = { 2, 6, 4, 8 };
16     list< int > values, otherValues;
17
18     values.push_front( 1 );
19     values.push_front( 2 );
20     values.push_back( 4 );
21     values.push_back( 3 );
22
23     cout << "values contains: ";
24     printList( values );
25     values.sort();
26     cout << "\nvalues after sorting contains: ";
27     printList( values );
28
29     otherValues.insert( otherValues.begin(), a, a + SIZE );

```

Fig. 20.17 Demonstrating Standard Library `list` class template (part 1 of 3).

```

30     cout << "\notherValues contains: ";
31     printList( otherValues );
32     values.splice( values.end(), otherValues );
33     cout << "\nAfter splice values contains: ";
34     printList( values );
35
36     values.sort();
37     cout << "\nvalues contains: ";
38     printList( values );
39     otherValues.insert( otherValues.begin(), a, a + SIZE );
40     otherValues.sort();
41     cout << "\notherValues contains: ";
42     printList( otherValues );
43     values.merge( otherValues );
44     cout << "\nAfter merge:\n  values contains: ";
45     printList( values );
46     cout << "\n  otherValues contains: ";
47     printList( otherValues );
48
49     values.pop_front();
50     values.pop_back(); // all sequence containers
51     cout << "\nAfter pop_front and pop_back values contains:\n";
52     printList( values );
53
54     values.unique();
55     cout << "\nAfter unique values contains: ";
56     printList( values );
57

```

```

58 // method swap is available in all containers
59 values.swap( otherValues );
60 cout << "\nAfter swap:\n  values contains: ";
61 printList( values );
62 cout << "\n  otherValues contains: ";
63 printList( otherValues );
64
65 values.assign( otherValues.begin(), otherValues.end() );
66 cout << "\nAfter assign values contains: ";
67 printList( values );
68
69 values.merge( otherValues );
70 cout << "\nvalues contains: ";
71 printList( values );
72 values.remove( 4 );
73 cout << "\nAfter remove( 4 ) values contains: ";
74 printList( values );
75 cout << endl;
76 return 0;
77 }
78

```

Fig. 20.17 Demonstrating Standard Library `list` class template (part 2 of 3).

```

79 template < class T >
80 void printList( const list< T > &listRef )
81 {
82     if ( listRef.empty() )
83         cout << "List is empty";
84     else {
85         ostream_iterator< T > output( cout, " " );
86         copy( listRef.begin(), listRef.end(), output );
87     }
88 }

```

```

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).

```

1 // Fig. 20.18: fig20_18.cpp
2 // Testing Standard Library class deque
3 #include <iostream>
4 #include <deque>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     deque< double > values;
12     ostream_iterator< double > output( cout, " " );
13
14     values.push_front( 2.2 );
15     values.push_front( 3.5 );
16     values.push_back( 1.1 );
17
18     cout << "values contains: ";
19
20     for ( int i = 0; i < values.size(); ++i )
21         cout << values[ i ] << ' ';
22
23     values.pop_front();
24     cout << "\nAfter pop_front values contains: ";
25     copy ( values.begin(), values.end(), output );
26
27     values[ 1 ] = 5.4;
28     cout << "\nAfter values[ 1 ] = 5.4 values contains: ";
29     copy ( values.begin(), values.end(), output );
30     cout << endl;
31     return 0;
32 }

```

```

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.

```

1 // Fig. 20.19: fig20_19.cpp
2 // Testing Standard Library class multiset
3 #include <iostream>
4 #include <set>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     const int SIZE = 10;
12     int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
13     typedef multiset< int, less< int > > ims;
14     ims intMultiset; // ims for "integer multiset"
15     ostream_iterator< int > output( cout, " " );
16
17     cout << "There are currently " << intMultiset.count( 15 )
18         << " values of 15 in the multiset\n";
19     intMultiset.insert( 15 );
20     intMultiset.insert( 15 );
21     cout << "After inserts, there are "

```

```

22     << intMultiset.count( 15 )
23     << " values of 15 in the multiset\n";
24
25     ims::const_iterator result;
26
27     result = intMultiset.find( 15 ); // find returns iterator
28
29     if ( result != intMultiset.end() ) // if iterator not at end
30         cout << "Found value 15\n"; // found search value 15
31
32     result = intMultiset.find( 20 );
33
34     if ( result == intMultiset.end() ) // will be true hence
35         cout << "Did not find value 20\n"; // did not find 20
36
37     intMultiset.insert( a, a + SIZE ); // add array a to multiset
38     cout << "After insert intMultiset contains:\n";
39     copy( intMultiset.begin(), intMultiset.end(), output );
40
41     cout << "\nLower bound of 22: "
42           << *( intMultiset.lower_bound( 22 ) );
43     cout << "\nUpper bound of 22: "
44           << *( intMultiset.upper_bound( 22 ) );
45
46     pair< ims::const_iterator, ims::const_iterator > p;
47
48     p = intMultiset.equal_range( 22 );
49     cout << "\nUsing equal_range of 22"
50           << "\n  Lower bound: " << *( p.first )
51           << "\n  Upper bound: " << *( p.second );

```

Fig. 20.19 Demonstrating Standard Library `multiset` class template (part 1 of 2).

```

52     cout << endl;
53     return 0;
54 }

```

```

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).

```

1 // Fig. 20.20: fig20_20.cpp
2 // Testing Standard Library class set
3 #include <iostream>
4 #include <set>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     typedef set< double, less< double > > double_set;
12     const int SIZE = 5;
13     double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
14     double_set doubleSet( a, a + SIZE );
15     ostream_iterator< double > output( cout, " " );
16
17     cout << "doubleSet contains: ";
18     copy( doubleSet.begin(), doubleSet.end(), output );
19
20     pair< double_set::const_iterator, bool > p;
21
22     p = doubleSet.insert( 13.8 ); // value not in set
23     cout << '\n' << *( p.first )
24         << ( p.second ? " was" : " was not" ) << " inserted";
25     cout << "\ndoubleSet contains: ";
26     copy( doubleSet.begin(), doubleSet.end(), output );
27
28     p = doubleSet.insert( 9.5 ); // value already in set
29     cout << '\n' << *( p.first )
30         << ( p.second ? " was" : " was not" ) << " inserted";
31     cout << "\ndoubleSet contains: ";
32     copy( doubleSet.begin(), doubleSet.end(), output );
33
34     cout << endl;
35     return 0;
36 }

```

```

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.

```

1 // Fig. 20.21: fig20_21.cpp
2 // Testing Standard Library class multimap
3 #include <iostream>
4 #include <map>
5
6 using namespace std;
7
8 int main()
9 {
10     typedef multimap< int, double, less< int > > mmid;
11     mmid pairs;
12

```



```

13     cout << "There are currently " << pairs.count( 15 )
14         << " pairs with key 15 in the multimap\n";
15     pairs.insert( mmid::value_type( 15, 2.7 ) );
16     pairs.insert( mmid::value_type( 15, 99.3 ) );
17     cout << "After inserts, there are "
18         << pairs.count( 15 )
19         << " pairs with key 15\n";
20     pairs.insert( mmid::value_type( 30, 111.11 ) );
21     pairs.insert( mmid::value_type( 10, 22.22 ) );
22     pairs.insert( mmid::value_type( 25, 33.333 ) );
23     pairs.insert( mmid::value_type( 20, 9.345 ) );
24     pairs.insert( mmid::value_type( 5, 77.54 ) );
25     cout << "Multimap pairs contains:\nKey\tValue\n";
26
27     for ( mmid::const_iterator iter = pairs.begin();
28         iter != pairs.end(); ++iter )
29         cout << iter->first << '\t'
30             << iter->second << '\n';
31
32     cout << endl;
33     return 0;
34 }

```

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).

```

1 // Fig. 20.22: fig20_22.cpp
2 // Testing Standard Library class map
3 #include <iostream>
4 #include <map>
5
6 using namespace std;
7
8 int main()
9 {
10     typedef map< int, double, less< int > > mid;
11     mid pairs;
12
13     pairs.insert( mid::value_type( 15, 2.7 ) );
14     pairs.insert( mid::value_type( 30, 111.11 ) );
15     pairs.insert( mid::value_type( 5, 1010.1 ) );
16     pairs.insert( mid::value_type( 10, 22.22 ) );
17     pairs.insert( mid::value_type( 25, 33.333 ) );
18     pairs.insert( mid::value_type( 5, 77.54 ) ); // dupe ignored

```

```

19     pairs.insert( mid::value_type( 20, 9.345 ) );
20     pairs.insert( mid::value_type( 15, 99.3 ) ); // dupe ignored
21     cout << "pairs contains:\nKey\tValue\n";
22
23     mid::const_iterator iter;
24
25     for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
26         cout << iter->first << '\t'
27             << iter->second << '\n';
28
29     pairs[ 25 ] = 9999.99; // change existing value for 25
30     pairs[ 40 ] = 8765.43; // insert new value for 40
31     cout << "\nAfter subscript operations, pairs contains:"
32         << "\nKey\tValue\n";
33
34     for ( iter = pairs.begin(); iter != pairs.end(); ++iter )
35         cout << iter->first << '\t'
36             << iter->second << '\n';
37
38     cout << endl;
39     return 0;
40 }

```

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).

```

1 // Fig. 20.23: fig20_23.cpp
2 // Testing Standard Library class stack
3 #include <iostream>
4 #include <stack>
5 #include <vector>
6 #include <list>
7
8 using namespace std;
9
10 template< class T >
11 void popElements( T &s );
12

```

```

13 int main()
14 {
15     stack< int > intDequeStack; // default is deque-based stack
16     stack< int, vector< int > > intVectorStack;
17     stack< int, list< int > > intListStack;
18
19     for ( int i = 0; i < 10; ++i ) {
20         intDequeStack.push( i );
21         intVectorStack.push( i );
22         intListStack.push( i );
23     }
24
25     cout << "Popping from intDequeStack: ";
26     popElements( intDequeStack );
27     cout << "\nPopping from intVectorStack: ";
28     popElements( intVectorStack );
29     cout << "\nPopping from intListStack: ";
30     popElements( intListStack );
31
32     cout << endl;
33     return 0;
34 }
35

```

Fig. 20.23 Demonstrating Standard Library **stack** adapter class (part 1 of 2).

```

36 template< class T >
37 void popElements( T &s )
38 {
39     while ( !s.empty() ) {
40         cout << s.top() << ' ';
41         s.pop();
42     }
43 }

```

```

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).

```
1 // Fig. 20.24: fig20_24.cpp
2 // Testing Standard Library adapter class template queue
3 #include <iostream>
4 #include <queue>
5
6 using namespace std;
7
8 int main()
9 {
10     queue< double > values;
11
12     values.push( 3.2 );
13     values.push( 9.8 );
14     values.push( 5.4 );
15
16     cout << "Popping from values: ";
17
18     while ( !values.empty() ) {
19         cout << values.front() << ' '; // does not remove
20         values.pop(); // removes element
21     }
22
23     cout << endl;
24     return 0;
25 }
```

```
Popping from values: 3.2 9.8 5.4
```

Fig. 20.24 Demonstrating Standard Library `queue` adapter class templates.

```
1 // Fig. 20.25: fig20_25.cpp
2 // Testing Standard Library class priority_queue
3 #include <iostream>
4 #include <queue>
5 #include <functional>
6
7 using namespace std;
8
9 int main()
10 {
11     priority_queue< double > priorities;
12
13     priorities.push( 3.2 );
14     priorities.push( 9.8 );
15     priorities.push( 5.4 );
16
17     cout << "Popping from priorities: ";
18
19     while ( !priorities.empty() ) {
20         cout << priorities.top() << ' ';
21         priorities.pop();
22     }
23
24     cout << endl;
25     return 0;
26 }
```

Popping from priorities: 9.8 5.4 3.2

Fig. 20.25 Demonstrating Standard Library `priority_queue` adapter class.

```

1 // Fig. 20.26: fig20_26.cpp
2 // Demonstrating fill, fill_n, generate, and generate_n
3 // Standard Library methods.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 char nextLetter();
11
12 int main()
13 {
14     vector< char > chars( 10 );
15     ostream_iterator< char > output( cout, " " );
16
17     fill( chars.begin(), chars.end(), '5' );
18     cout << "Vector chars after filling with 5s:\n";
19     copy( chars.begin(), chars.end(), output );
20
21     fill_n( chars.begin(), 5, 'A' );
22     cout << "\nVector chars after filling five elements"
23           << " with As:\n";
24     copy( chars.begin(), chars.end(), output );
25
26     generate( chars.begin(), chars.end(), nextLetter );
27     cout << "\nVector chars after generating letters A-J:\n";
28     copy( chars.begin(), chars.end(), output );

```

Fig. 20.26 Demonstrating Standard Library functions `fill`, `fill_n`, `generate` and `generate_n` (part 1 of 2).

```

29
30     generate_n( chars.begin(), 5, nextLetter );
31     cout << "\nVector chars after generating K-O for the"
32           << " first five elements:\n";
33     copy( chars.begin(), chars.end(), output );
34
35     cout << endl;
36     return 0;
37 }
38
39 char nextLetter()
40 {
41     static char letter = 'A';
42     return letter++;
43 }

```

```

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).

```

1 // Fig. 20.27: fig20_27.cpp
2 // Demonstrates Standard Library functions equal,
3 // mismatch, lexicographical_compare.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 10;
13     int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14     int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
15     vector< int > v1( a1, a1 + SIZE ),
16                 v2( a1, a1 + SIZE ),
17                 v3( a2, a2 + SIZE );
18     ostream_iterator< int > output( cout, " " );
19

```

Fig. 20.27 Demonstrating Standard Library functions `equal`, `mismatch` and `lexicographical_compare` (part 1 of 2).

```

20     cout << "Vector v1 contains: ";
21     copy( v1.begin(), v1.end(), output );
22     cout << "\nVector v2 contains: ";
23     copy( v2.begin(), v2.end(), output );
24     cout << "\nVector v3 contains: ";
25     copy( v3.begin(), v3.end(), output );
26
27     bool result = equal( v1.begin(), v1.end(), v2.begin() );
28     cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
29         << " equal to vector v2.\n";
30
31     result = equal( v1.begin(), v1.end(), v3.begin() );
32     cout << "Vector v1 " << ( result ? "is" : "is not" )
33         << " equal to vector v3.\n";
34
35     pair< vector< int >::iterator,
36         vector< int >::iterator > location;
37     location = mismatch( v1.begin(), v1.end(), v3.begin() );
38     cout << "\nThere is a mismatch between v1 and v3 at "

```

```

39         << "location " << ( location.first - v1.begin() )
40         << "\nwhere v1 contains " << *location.first
41         << " and v3 contains " << *location.second
42         << "\n\n";
43
44     char c1[ SIZE ] = "HELLO", c2[ SIZE ] = "BYE BYE";
45
46     result =
47         lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
48     cout << c1
49         << ( result ? " is less than " : " is greater than " )
50         << c2;
51
52     cout << endl;
53     return 0;
54 }

```

```

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).

```

1 // Fig. 20.28: fig20_28.cpp
2 // Demonstrates Standard Library functions remove, remove_if
3 // remove_copy and remove_copy_if
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 bool greater9( int );
11
12 int main()
13 {
14     const int SIZE = 10;
15     int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
16     ostream_iterator< int > output( cout, " " );
17
18     // Remove 10 from v
19     vector< int > v( a, a + SIZE );
20     vector< int >::iterator newLastElement;
21     cout << "Vector v before removing all 10s:\n";
22     copy( v.begin(), v.end(), output );
23     newLastElement = remove( v.begin(), v.end(), 10 );
24     cout << "\nVector v after removing all 10s:\n";
25     copy( v.begin(), newLastElement, output );
26
27     // Copy from v2 to c, removing 10s

```

```

28     vector< int > v2( a, a + SIZE );
29     vector< int > c( SIZE, 0 );
30     cout << "\n\nVector v2 before removing all 10s "
31           << "and copying:\n";
32     copy( v2.begin(), v2.end(), output );
33     remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
34     cout << "\nVector c after removing all 10s from v2:\n";
35     copy( c.begin(), c.end(), output );
36

```

Fig. 20.28 Demonstrating Standard Library functions `remove`, `remove_if`, `remove_copy` and `remove_copy_if` (part 1 of 3).

```

37     // Remove elements greater than 9 from v3
38     vector< int > v3( a, a + SIZE );
39     cout << "\n\nVector v3 before removing all elements"
40           << "\ngreater than 9:\n";
41     copy( v3.begin(), v3.end(), output );
42     newLastElement = remove_if( v3.begin(), v3.end(),
43                                greater9 );
44     cout << "\nVector v3 after removing all elements"
45           << "\ngreater than 9:\n";
46     copy( v3.begin(), newLastElement, output );
47
48     // Copy elements from v4 to c,
49     // removing elements greater than 9
50     vector< int > v4( a, a + SIZE );
51     vector< int > c2( SIZE, 0 );
52     cout << "\n\nVector v4 before removing all elements"
53           << "\ngreater than 9 and copying:\n";
54     copy( v4.begin(), v4.end(), output );
55     remove_copy_if( v4.begin(), v4.end(),
56                   c2.begin(), greater9 );
57     cout << "\nVector c2 after removing all elements"
58           << "\ngreater than 9 from v4:\n";
59     copy( c2.begin(), c2.end(), output );
60
61     cout << endl;
62     return 0;
63 }
64
65 bool greater9( int x )
66 {
67     return x > 9;
68 }

```

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).

```

1 // Fig. 20.29: fig20_29.cpp
2 // Demonstrates Standard Library functions replace, replace_if
3 // replace_copy and replace_copy_if
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 bool greater9( int );
11
12 int main()
13 {
14     const int SIZE = 10;
15     int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
16     ostream_iterator< int > output( cout, " " );
17
18     // Replace 10s in v1 with 100
19     vector< int > v1( a, a + SIZE );
20     cout << "Vector v1 before replacing all 10s:\n";
21     copy( v1.begin(), v1.end(), output );
22     replace( v1.begin(), v1.end(), 10, 100 );
23     cout << "\nVector v1 after replacing all 10s with 100s:\n";
24     copy( v1.begin(), v1.end(), output );
25

```

Fig. 20.29 Demonstrating Standard Library functions `replace`, `replace_if`, `replace_copy` and `replace_copy_if` (part 1 of 3).

```

26 // copy from v2 to c1, replacing 10s with 100s
27 vector< int > v2( a, a + SIZE );

```

```

28     vector< int > c1( SIZE );
29     cout << "\n\nVector v2 before replacing all 10s "
30         << "and copying:\n";
31     copy( v2.begin(), v2.end(), output );
32     replace_copy( v2.begin(), v2.end(),
33                 c1.begin(), 10, 100 );
34     cout << "\n\nVector c1 after replacing all 10s in v2:\n";
35     copy( c1.begin(), c1.end(), output );
36
37     // Replace values greater than 9 in v3 with 100
38     vector< int > v3( a, a + SIZE );
39     cout << "\n\nVector v3 before replacing values greater"
40         << " than 9:\n";
41     copy( v3.begin(), v3.end(), output );
42     replace_if( v3.begin(), v3.end(), greater9, 100 );
43     cout << "\n\nVector v3 after replacing all values greater"
44         << "\n than 9 with 100s:\n";
45     copy( v3.begin(), v3.end(), output );
46
47     // Copy v4 to c2, replacing elements greater than 9 with 100
48     vector< int > v4( a, a + SIZE );
49     vector< int > c2( SIZE );
50     cout << "\n\nVector v4 before replacing all values greater"
51         << "\n than 9 and copying:\n";
52     copy( v4.begin(), v4.end(), output );
53     replace_copy_if( v4.begin(), v4.end(), c2.begin(),
54                    greater9, 100 );
55     cout << "\n\nVector c2 after replacing all values greater"
56         << "\n than 9 in v4:\n";
57     copy( c2.begin(), c2.end(), output );
58
59     cout << endl;
60     return 0;
61 }
62
63 bool greater9( int x )
64 {
65     return x > 9;
66 }

```

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).

```

1 // Fig. 20.30: fig20_30.cpp
2 // Examples of mathematical algorithms in the Standard Library.
3 #include <iostream>
4 #include <algorithm>
5 #include <numeric> // accumulate is defined here
6 #include <vector>
7
8 using namespace std;
9
10 bool greater9( int );
11 void outputSquare( int );
12 int calculateCube( int );
13
14 int main()
15 {
16     const int SIZE = 10;
17     int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
18     vector< int > v( a1, a1 + SIZE );
19     ostream_iterator< int > output( cout, " " );
20
21     cout << "Vector v before random_shuffle: ";
22     copy( v.begin(), v.end(), output );
23     random_shuffle( v.begin(), v.end() );
24     cout << "\nVector v after random_shuffle: ";
25     copy( v.begin(), v.end(), output );
26
27     int a2[] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
28     vector< int > v2( a2, a2 + SIZE );

```

Fig. 20.30 Demonstrating some mathematical algorithms of the Standard Library (part 1 of 3).

```

29     cout << "\n\nVector v2 contains: ";

```

```

30     copy( v2.begin(), v2.end(), output );
31     int result = count( v2.begin(), v2.end(), 8 );
32     cout << "\nNumber of elements matching 8: " << result;
33
34     result = count_if( v2.begin(), v2.end(), greater9 );
35     cout << "\nNumber of elements greater than 9: " << result;
36
37     cout << "\n\nMinimum element in Vector v2 is: "
38           << *( min_element( v2.begin(), v2.end() ) );
39
40     cout << "\n\nMaximum element in Vector v2 is: "
41           << *( max_element( v2.begin(), v2.end() ) );
42
43     cout << "\n\nThe total of the elements in Vector v is: "
44           << accumulate( v.begin(), v.end(), 0 );
45
46     cout << "\n\nThe square of every integer in Vector v is:\n";
47     for_each( v.begin(), v.end(), outputSquare );
48
49     vector< int > cubes( SIZE );
50     transform( v.begin(), v.end(), cubes.begin(),
51              calculateCube );
52     cout << "\n\nThe cube of every integer in Vector v is:\n";
53     copy( cubes.begin(), cubes.end(), output );
54
55     cout << endl;
56     return 0;
57 }
58
59 bool greater9( int value ) { return value > 9; }
60
61 void outputSquare( int value ) { cout << value * value << ' '; }
62
63 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).

```

1 // Fig. 20.31: fig20_31.cpp
2 // Demonstrates search and sort capabilities.
3 #include <iostream>
4 #include <algorithm>
5 #include <vector>
6
7 using namespace std;
8
9 bool greater10( int value );
10
11 int main()
12 {
13     const int SIZE = 10;
14     int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
15     vector< int > v( a, a + SIZE );
16     ostream_iterator< int > output( cout, " " );
17
18     cout << "Vector v contains: ";
19     copy( v.begin(), v.end(), output );
20
21     vector< int >::iterator location;
22     location = find( v.begin(), v.end(), 16 );
23
24     if ( location != v.end() )
25         cout << "\n\nFound 16 at location "
26             << ( location - v.begin() );
27     else
28         cout << "\n\n16 not found";
29
30     location = find( v.begin(), v.end(), 100 );
31
32     if ( location != v.end() )
33         cout << "\n\nFound 100 at location "
34             << ( location - v.begin() );
35     else
36         cout << "\n\n100 not found";
37
38     location = find_if( v.begin(), v.end(), greater10 );
39
40     if ( location != v.end() )
41         cout << "\n\nThe first value greater than 10 is "
42             << *location << "\n\nfound at location "
43             << ( location - v.begin() );
44     else
45         cout << "\n\nNo values greater than 10 were found";
46
47     sort( v.begin(), v.end() );
48     cout << "\n\nVector v after sort: ";
49     copy( v.begin(), v.end(), output );
50

```

Fig. 20.31 Basic searching and sorting algorithms of the Standard Library (part 1 of 2).

```

51     if ( binary_search( v.begin(), v.end(), 13 ) )
52         cout << "\n\n13 was found in v";
53     else
54         cout << "\n\n13 was not found in v";
55
56     if ( binary_search( v.begin(), v.end(), 100 ) )
57         cout << "\n\n100 was found in v";
58     else
59         cout << "\n\n100 was not found in v";
60

```

```

61     cout << endl;
62     return 0;
63 }
64
65 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).

```

1 // Fig. 20.32: fig20_32.cpp
2 // Demonstrates iter_swap, swap and swap_ranges.
3 #include <iostream>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10     const int SIZE = 10;
11     int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
12     ostream_iterator< int > output( cout, " " );
13
14     cout << "Array a contains:\n";
15     copy( a, a + SIZE, output );

```

Fig. 20.32 Demonstrating `swap`, `iter_swap` and `swap_ranges` (part 1 of 2).

```

16     swap( a[ 0 ], a[ 1 ] );
17     cout << "\nArray a after swapping a[0] and a[1] "
18         << "using swap:\n";
19     copy( a, a + SIZE, output );
20
21     iter_swap( &a[ 0 ], &a[ 1 ] );
22     cout << "\nArray a after swapping a[0] and a[1] "
23         << "using iter_swap:\n";
24     copy( a, a + SIZE, output );
25
26     swap_ranges( a, a + 5, a + 5 );
27     cout << "\nArray a after swapping the first five elements\n"
28         << "with the last five elements:\n";
29     copy( a, a + SIZE, output );
30
31     cout << endl;
32     return 0;
33 }
34

```

```

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).

```

1 // Fig. 20.33: fig20_33.cpp
2 // Demonstrates miscellaneous functions: copy_backward, merge,
3 // unique and reverse.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 5;
13     int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
14     int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
15     vector< int > v1( a1, a1 + SIZE );
16     vector< int > v2( a2, a2 + SIZE );
17
18     ostream_iterator< int > output( cout, " " );
19
20     cout << "Vector v1 contains: ";
21     copy( v1.begin(), v1.end(), output );
22     cout << "\nVector v2 contains: ";
23     copy( v2.begin(), v2.end(), output );
24
25     vector< int > results( v1.size() );
26     copy_backward( v1.begin(), v1.end(), results.end() );
27     cout << "\n\nAfter copy_backward, results contains: ";
28     copy( results.begin(), results.end(), output );
29
30     vector< int > results2( v1.size() * v2.size() );
31     merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
32           results2.begin() );
33     cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
34     copy( results2.begin(), results2.end(), output );
35
36     vector< int >::iterator endLocation;
37     endLocation = unique( results2.begin(), results2.end() );

```

Fig. 20.33 Demonstrating `copy_backward`, `merge`, `unique` and `reverse` (part 2 of 2).

```

38     cout << "\n\nAfter unique results2 contains:\n";
39     copy( results2.begin(), endLocation, output );
40
41     cout << "\n\nVector v1 after reverse: ";
42     reverse( v1.begin(), v1.end() );

```

```

43     copy( v1.begin(), v1.end(), output );
44
45     cout << endl;
46     return 0;
47 }

```

```

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).

```

1 // Fig. 20.34: fig20_34.cpp
2 // Demonstrates miscellaneous functions: inplace_merge,
3 // reverse_copy, and unique_copy.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7 #include <iterator>
8
9 using namespace std;
10
11 int main()
12 {
13     const int SIZE = 10;
14     int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
15     vector< int > v1( a1, a1 + SIZE );
16
17     ostream_iterator< int > output( cout, " " );
18
19     cout << "Vector v1 contains: ";
20     copy( v1.begin(), v1.end(), output );
21
22     inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
23     cout << "\nAfter inplace_merge, v1 contains: ";
24     copy( v1.begin(), v1.end(), output );
25
26     vector< int > results1;
27     unique_copy( v1.begin(), v1.end(),
28                 back_inserter( results1 ) );
29     cout << "\nAfter unique_copy results1 contains: ";
30     copy( results1.begin(), results1.end(), output );
31
32     vector< int > results2;
33     cout << "\nAfter reverse_copy, results2 contains: ";
34     reverse_copy( v1.begin(), v1.end(),
35                  back_inserter( results2 ) );
36     copy( results2.begin(), results2.end(), output );
37
38     cout << endl;
39     return 0;
40 }

```



```

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`.

```

1 // Fig. 20.35: fig20_35.cpp
2 // Demonstrates includes, set_difference, set_intersection,
3 // set_symmetric_difference and set_union.
4 #include <iostream>
5 #include <algorithm>
6
7 using namespace std;
8
9 int main()
10 {
11     const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
12     int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13     int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
14     int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
15     ostream_iterator< int > output( cout, " " );
16
17     cout << "a1 contains: ";
18     copy( a1, a1 + SIZE1, output );
19     cout << "\na2 contains: ";
20     copy( a2, a2 + SIZE2, output );
21     cout << "\na3 contains: ";
22     copy( a3, a3 + SIZE2, output );
23
24     if ( includes( a1, a1 + SIZE1, a2, a2 + SIZE2 ) )
25         cout << "\na1 includes a2";
26     else
27         cout << "\na1 does not include a2";
28
29     if ( includes( a1, a1 + SIZE1, a3, a3 + SIZE2 ) )
30         cout << "\na1 includes a3";
31     else
32         cout << "\na1 does not include a3";
33
34     int difference[ SIZE1 ];
35     int *ptr = set_difference( a1, a1 + SIZE1, a2, a2 + SIZE2,
36                             difference );
37     cout << "\nset_difference of a1 and a2 is: ";
38     copy( difference, ptr, output );
39
40     int intersection[ SIZE1 ];
41     ptr = set_intersection( a1, a1 + SIZE1, a2, a2 + SIZE2,
42                            intersection );

```

Fig. 20.35 Demonstrating `set` operations of the Standard Library (part 1 of 2).

```

43     cout << "\nset_intersection of a1 and a2 is: ";
44     copy( intersection, ptr, output );
45
46     int symmetric_difference[ SIZE1 ];
47     ptr = set_symmetric_difference( a1, a1 + SIZE1,

```

```

48         a2, a2 + SIZE2, symmetric_difference );
49     cout << "\nset_symmetric_difference of a1 and a2 is: ";
50     copy( symmetric_difference, ptr, output );
51
52     int unionSet[ SIZE3 ];
53     ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
54     cout << "\nset_union of a1 and a3 is: ";
55     copy( unionSet, ptr, output );
56     cout << endl;
57     return 0;
58 }

```

```

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).

```

1 // Fig. 20.36: fig20_36.cpp
2 // Demonstrates lower_bound, upper_bound and equal_range for
3 // a sorted sequence of values.
4 #include <iostream>
5 #include <algorithm>
6 #include <vector>
7
8 using namespace std;
9
10 int main()
11 {
12     const int SIZE = 10;
13     int a1[] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
14     vector< int > v( a1, a1 + SIZE );
15     ostream_iterator< int > output( cout, " " );
16
17     cout << "Vector v contains:\n";
18     copy( v.begin(), v.end(), output );
19
20     vector< int >::iterator lower;
21     lower = lower_bound( v.begin(), v.end(), 6 );
22     cout << "\n\nLower bound of 6 is element "
23         << ( lower - v.begin() ) << " of vector v";
24
25     vector< int >::iterator upper;
26     upper = upper_bound( v.begin(), v.end(), 6 );
27     cout << "\n\nUpper bound of 6 is element "
28         << ( upper - v.begin() ) << " of vector v";
29
30     pair< vector< int >::iterator, vector< int >::iterator > eq;
31     eq = equal_range( v.begin(), v.end(), 6 );
32     cout << "\n\nUsing equal_range:\n"
33         << "    Lower bound of 6 is element "
34         << ( eq.first - v.begin() ) << " of vector v";
35     cout << "\n    Upper bound of 6 is element "
36         << ( eq.second - v.begin() ) << " of vector v";
37

```

```

38 cout << "\n\nUse lower_bound to locate the first point\n"
39     << "at which 5 can be inserted in order";
40 lower = lower_bound( v.begin(), v.end(), 5 );
41 cout << "\n  Lower bound of 5 is element "
42     << ( lower - v.begin() ) << " of vector v";
43
44 cout << "\n\nUse upper_bound to locate the last point\n"
45     << "at which 7 can be inserted in order";
46 upper = upper_bound( v.begin(), v.end(), 7 );
47 cout << "\n  Upper bound of 7 is element "
48     << ( upper - v.begin() ) << " of vector v";
49

```

Fig. 20.36 Demonstrating `lower_bound`, `upper_bound` and `equal_range` (part 1 of 2).

```

50 cout << "\n\nUse equal_range to locate the first and\n"
51     << "last point at which 5 can be inserted in order";
52 eq = equal_range( v.begin(), v.end(), 5 );
53 cout << "\n  Lower bound of 5 is element "
54     << ( eq.first - v.begin() ) << " of vector v";
55 cout << "\n  Upper bound of 5 is element "
56     << ( eq.second - v.begin() ) << " of vector v"
57     << endl;
58 return 0;
59 }

```

```

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).

```

1 // Fig. 20.37: fig20_37.cpp
2 // Demonstrating push_heap, pop_heap, make_heap and sort_heap.
3 #include <iostream>
4 #include <algorithm>
5 #include <vector>
6
7 using namespace std;
8
9 int main()
10 {
11     const int SIZE = 10;
12     int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
13     int i;

```

Fig. 20.37 Using Standard Library functions to perform a heapsort (part 1 of 3).

```

14     vector< int > v( a, a + SIZE ), v2;
15     ostream_iterator< int > output( cout, " " );
16
17     cout << "Vector v before make_heap:\n";
18     copy( v.begin(), v.end(), output );
19     make_heap( v.begin(), v.end() );
20     cout << "\nVector v after make_heap:\n";
21     copy( v.begin(), v.end(), output );
22     sort_heap( v.begin(), v.end() );
23     cout << "\nVector v after sort_heap:\n";
24     copy( v.begin(), v.end(), output );
25
26     // perform the heapsort with push_heap and pop_heap
27     cout << "\n\nArray a contains: ";
28     copy( a, a + SIZE, output );
29
30     for ( i = 0; i < SIZE; ++i ) {
31         v2.push_back( a[ i ] );
32         push_heap( v2.begin(), v2.end() );
33         cout << "\nv2 after push_heap(a[" << i << "]): ";
34         copy( v2.begin(), v2.end(), output );
35     }
36
37     for ( i = 0; i < v2.size(); ++i ) {
38         cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
39         pop_heap( v2.begin(), v2.end() - i );
40         copy( v2.begin(), v2.end(), output );
41     }
42
43     cout << endl;
44     return 0;
45 }

```

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).

```

1 // Fig. 20.38: fig20_38.cpp
2 // Demonstrating min and max
3 #include <iostream>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "The minimum of 12 and 7 is: " << min( 12, 7 );
11     cout << "\nThe maximum of 12 and 7 is: " << max( 12, 7 );
12     cout << "\nThe minimum of 'G' and 'Z' is: "
13         << min( 'G', 'Z' );
14     cout << "\nThe maximum of 'G' and 'Z' is: "
15         << max( 'G', 'Z' ) << endl;
16     return 0;
17 }

```

```

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
<code>adjacent_difference</code>	Beginning with the second element in a sequence, calculate the difference (using operator <code>-</code>) 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.
<code>inner_product</code>	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.
<code>partial_sum</code>	Calculate a running total (using operator <code>+</code>) 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.
<code>nth_element</code>	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.
<code>partition</code>	This function is similar to <code>nth_element</code> ; however, it requires less powerful bidirectional iterators, so it is more flexible than <code>nth_element</code> . Function <code>partition</code> 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 <code>true</code> are to the left (toward the beginning of the sequence) of all elements for which the predicate is <code>false</code> . A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns <code>false</code> .

Fig. 20.39 Algorithms not covered in this chapter.

Algorithm	Description
<code>stable_partition</code>	This function is similar to <code>partition</code> except that elements for which the predicate function returns <code>true</code> are maintained in their original order and elements for which the <code>predicate</code> function returns <code>false</code> are maintained in their original order.
<code>next_permutation</code>	Next lexicographical permutation of a sequence.
<code>prev_permutation</code>	Previous lexicographical permutation of a sequence.
<code>rotate</code>	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.
<code>rotate_copy</code>	This function is identical to <code>rotate</code> 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.
<code>adjacent_find</code>	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 <code>end</code> of the sequence.
<code>partial_sort</code>	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 <code><</code> (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.
<code>partial_sort_copy</code>	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 <code><</code> (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.
<code>stable_sort</code>	The function is similar to <code>sort</code> except that all equal elements are maintained in their original order.

Fig. 20.39 Algorithms not covered in this chapter.

```

1 // Fig. 20.40: fig20_40.cpp
2 // Using a bitset to demonstrate the Sieve of Eratosthenes.
3 #include <iostream>
4 #include <iomanip>
5 #include <bitset>
6 #include <cmath>
7
8 using namespace std;
9
10 int main()
11 {
12     const int size = 1024;
13     int i, value, counter;
14     bitset< size > sieve;
15
16     sieve.flip();
17

```

Fig. 20.40 Demonstrating class `bitset` and the Sieve of Eratosthenes (part 1 of 3).

```

18     // perform Sieve of Eratosthenes
19     int finalBit = sqrt( sieve.size() ) + 1;
20
21     for ( i = 2; i < finalBit; ++i )
22         if ( sieve.test( i ) )
23             for ( int j = 2 * i; j < size; j += i )
24                 sieve.reset( j );
25
26     cout << "The prime numbers in the range 2 to 1023 are:\n";
27
28     for ( i = 2, counter = 0; i < size; ++i )
29         if ( sieve.test( i ) ) {
30             cout << setw( 5 ) << i;
31
32             if ( ++counter % 12 == 0 )
33                 cout << '\n';
34         }
35
36     cout << endl;
37
38     // get a value from the user to determine if it is prime
39     cout << "\nEnter a value from 1 to 1023 (-1 to end): ";
40     cin >> value;
41
42     while ( value != -1 ) {
43         if ( sieve[ value ] )
44             cout << value << " is a prime number\n";
45         else
46             cout << value << " is not a prime number\n";
47
48         cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
49         cin >> value;
50     }
51
52     return 0;
53 }

```

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
<code>divides< T ></code>	arithmetic
<code>equal_to< T ></code>	relational
<code>greater< T ></code>	relational
<code>greater_equal< T ></code>	relational
<code>less< T ></code>	relational
<code>less_equal< T ></code>	relational
<code>logical_and< T ></code>	logical
<code>logical_not< T ></code>	logical
<code>logical_or< T ></code>	logical
<code>minus< T ></code>	arithmetic
<code>modulus< T ></code>	arithmetic
<code>negate< T ></code>	arithmetic
<code>not_equal_to< T ></code>	relational
<code>plus< T ></code>	arithmetic
<code>multiplies< T ></code>	arithmetic

Fig. 20.41 Function objects in the Standard Library .

```

1 // Fig. 20.42: fig20_42.cpp
2 // Demonstrating function objects.
3 #include <iostream>
4 #include <vector>
5 #include <algorithm>
6 #include <numeric>
7 #include <functional>
8
9 using namespace std;
10
11 // binary function adds the square of its second argument and
12 // the running total in its first argument and
13 // returns the sum
14 int sumSquares( int total, int value )
15     { return total + value * value; }

```

Fig. 20.42 Demonstrating a binary function object (part 1 of 2).

```

16
17 // binary function class template which defines an overloaded
18 // operator() that function adds the square of its second
19 // argument and the running total in its first argument and
20 // returns the sum
21 template< class T >
22 class SumSquaresClass : public binary_function< T, T, T >
23 {
24 public:
25     const T &operator()( const T &total, const T &value )
26         { return total + value * value; }
27 };
28
29 int main()
30 {
31     const int SIZE = 10;
32     int a1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
33     vector< int > v( a1, a1 + SIZE );
34     ostream_iterator< int > output( cout, " " );
35     int result = 0;
36
37     cout << "vector v contains:\n";
38     copy( v.begin(), v.end(), output );
39     result = accumulate( v.begin(), v.end(), 0, sumSquares );
40     cout << "\n\nSum of squares of elements in vector v using "
41         << "binary\nfunction sumSquares: " << result;
42
43     result = accumulate( v.begin(), v.end(), 0,
44                         SumSquaresClass< int >() );
45     cout << "\n\nSum of squares of elements in vector v using "
46         << "binary\nfunction object of type "
47         << "SumSquaresClass< int >: " << result << endl;
48     return 0;
49 }

```

```
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** Demonstrating the fundamental data type **bool**.
- Fig. 21.2** Demonstrating operator **static_cast**.
- Fig. 21.3** Demonstrating the **const_cast** operator.
- Fig. 21.4** Demonstrating operator **reinterpret_cast**.
- Fig. 21.5** Demonstrating the use of **namespaces**.
- Fig. 21.6** Demonstrating **typeid**.
- Fig. 21.7** Demonstrating **dynamic_cast**.
- Fig. 21.8** Operator Keywords as alternatives to operator symbols.
- Fig. 21.9** Demonstrating the use of the operator keyword **.**
- Fig. 21.10** Single-argument constructors and implicit conversions
- Fig. 21.11** Demonstrating an **explicit** constructor
- Fig. 21.12** Demonstrating a **mutable** data member
- Fig. 21.13** Demonstrating the **.*** and **->*** operators.
- Fig. 21.14** Multiple inheritance to form class **iostream**.
- Fig. 21.15** Attempting to call a multiply inherited function polymorphically.
- Fig. 21.16** Using **virtual** base classes.

```
1 // Fig. 21.1: fig21_01.cpp
2 // Demonstrating data type bool.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
```

Fig. 21.1 Demonstrating the fundamental data type `bool` (part 1 of 2).

```
7 int main()
8 {
9     bool boolean = false;
10    int x = 0;
11
12    cout << "boolean is " << boolean
13         << "\nEnter an integer: ";
14    cin >> x;
15
16    cout << "integer " << x << " is"
17         << ( x ? " nonzero " : " zero " )
18         << "and interpreted as ";
19
20    if ( x )
21        cout << "true\n";
22    else
23        cout << "false\n";
24
25    boolean = true;
26    cout << "boolean is " << boolean;
27    cout << "\nboolean output with boolalpha manipulator is "
28         << boolalpha << boolean << endl;
29
30    return 0;
31 }
```

```
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).

```

1 // Fig. 21.2: fig21_02.cpp
2 // Demonstrating the static_cast operator.
3 #include <iostream.h>
4
5 class BaseClass {
6 public:
7     void f( void ) const { cout << "BASE\n"; }
8 };
9
10 class DerivedClass: public BaseClass {
11 public:
12     void f( void ) const { cout << "DERIVED\n"; }
13 };
14
15 void test( BaseClass * );
16
17 int main()
18 {
19     // use static_cast for a conversion
20     double d = 8.22;
21     int x = static_cast< int >( d );
22
23     cout << "d is " << d << "\nx is " << x << endl;
24
25     BaseClass base; // instantiate base object
26     test( &base ); // call test
27
28     return 0;
29 }
30
31 void test( BaseClass *basePtr )
32 {
33     DerivedClass *derivedPtr;
34
35     // cast base class pointer into derived class pointer
36     derivedPtr = static_cast< DerivedClass * >( basePtr );
37     derivedPtr->f(); // invoke DerivedClass function f
38 }

```

```

d is 8.22
x is 8
DERIVED

```

Fig. 21.2 Demonstrating operator `static_cast`.

```

1 // Fig. 21.3: fig21_03.cpp
2 // Demonstrating the const_cast operator.
3 #include <iostream.h>
4
5 class ConstCastTest {
6 public:
7     void setNumber( int );
8     int getNumber() const;
9     void printNumber() const;
10 private:
11     int number;
12 };
13
14 void ConstCastTest::setNumber( int num ) { number = num; }

```

```

15
16 int ConstCastTest::getNumber() const { return number; }
17
18 void ConstCastTest::printNumber() const
19 {
20     cout << "\nNumber after modification: ";
21
22     // the expression number-- would generate compile error
23     // undo const-ness to allow modification
24     const_cast< ConstCastTest * >( this )->number--;
25
26     cout << number << endl;
27 }
28

```

Fig. 21.3 Demonstrating the `const_cast` operator (part 1 of 2).

```

29 int main()
30 {
31     ConstCastTest x;
32     x.setNumber( 8 ); // set private data number to 8
33
34     cout << "Initial value of number: " << x.getNumber();
35
36     x.printNumber();
37     return 0;
38 }

```

```

Initial value of number: 8
Number after modification: 7

```

Fig. 21.3 Demonstrating the `const_cast` operator (part 2 of 2).

```

1 // Fig. 21.4: fig21_04.cpp
2 // Demonstrating reinterpret_cast operator.
3 #include <iostream.h>
4
5 int main()
6 {
7     unsigned x = 22, *unsignedPtr;
8     void *voidPtr = &x;
9     char *charPtr = "C++";

```

Fig. 21.4 Demonstrating operator `reinterpret_cast` (part 1 of 2).

```

10
11 // cast from void * to unsigned *
12 unsignedPtr = reinterpret_cast< unsigned * >( voidPtr );
13
14 cout << "*unsignedPtr is " << *unsignedPtr
15     << "\ncharPtr is " << charPtr;
16
17 // use reinterpret_cast to cast a char * pointer to unsigned
18 cout << "\nchar * to unsigned results in: "
19     << ( x = reinterpret_cast< unsigned >( charPtr ) );
20
21 // cast unsigned back to char *
22 cout << "\nunsigned to char * results in: "

```

```

23         << reinterpret_cast< char * >( x ) << endl;
24
25     return 0;
26 }

```

```

*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).

```

1 // Fig. 21.5: fig21_05.cpp
2 // Demonstrating namespaces.
3 #include <iostream>
4 using namespace std; // use std namespace
5
6 int myInt = 98; // global variable
7
8 namespace Example {
9     const double PI = 3.14159;
10    const double E = 2.71828;
11    int myInt = 8;
12    void printValues();
13
14    namespace Inner { // nested namespace
15        enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
16    }
17 }
18
19 namespace { // unnamed namespace
20     double d = 88.22;
21 }
22
23 int main()
24 {
25     // output value d of unnamed namespace
26     cout << "d = " << d;
27
28     // output global variable
29     cout << "\n(global) myInt = " << myInt;
30
31     // output values of Example namespace
32     cout << "\nPI = " << Example::PI << "\nE = "
33         << Example::E << "\nmyInt = "
34         << Example::myInt << "\nFISCAL3 = "
35         << Example::Inner::FISCAL3 << endl;
36
37     Example::printValues(); // invoke printValues function
38
39     return 0;
40 }
41
42 void Example::printValues()
43 {
44     cout << "\n\nIn printValues:\n" << "myInt = "
45         << myInt << "\nPI = " << PI << "\nE = "
46         << E << "\nd = " << d << "\n(global) myInt = "
47         << ::myInt << "\nFISCAL3 = "
48         << Inner::FISCAL3 << endl;

```



```
49 }
```

Fig. 21.5 Demonstrating the use of **namespaces** (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 **namespaces** (part 2 of 2).

```

1 // Fig. 21.6: fig21_06.cpp
2 // Demonstrating RTTI capability typeid.
3 #include <iostream.h>
4 #include <typeinfo.h>
5
6 template < typename T >
7 T maximum( T value1, T value2, T value3 )
8 {
9     T max = value1;
10
11     if ( value2 > max )
12         max = value2;
13
14     if ( value3 > max )
15         max = value3;
16
17     // get the name of the type (i.e., int or double)
18     const char *dataType = typeid( T ).name();
19
20     cout << dataType << "s were compared.\nLargest "
21         << dataType << " is ";
22
23     return max;
24 }
25
26 int main()
27 {
28     int a = 8, b = 88, c = 22;
29     double d = 95.96, e = 78.59, f = 83.89;

```

Fig. 21.6 Demonstrating **typeid** (part 1 of 2).

```

30
31     cout << maximum( a, b, c ) << "\n";
32     cout << maximum( d, e, f ) << endl;

```

```

33
34     return 0;
35 }

```

```

ints were compared.
Largest int is 88
doubles were compared.
Largest double is 95.96

```

Fig. 21.6 Demonstrating `typeid` (part 2 of 2).

```

1 // Fig. 21.7: fig21_07.cpp
2 // Demonstrating dynamic_cast.
3 #include <iostream.h>
4
5 const double PI = 3.14159;
6

```

Fig. 21.7 Demonstrating `dynamic_cast` (part 1 of 3).

```

7 class Shape {
8     public:
9         virtual double area() const { return 0.0; }
10 };
11
12 class Circle: public Shape {
13     public:
14         Circle( int r = 1 ) { radius = r; }
15
16         virtual double area() const
17         {
18             return PI * radius * radius;
19         };
20     protected:
21         int radius;
22 };
23
24 class Cylinder: public Circle {
25     public:
26         Cylinder( int h = 1 ) { height = h; }
27
28         virtual double area() const
29         {
30             return 2 * PI * radius * height +
31                 2 * Circle::area();
32         }
33     private:
34         int height;
35 };
36
37 void outputShapeArea( const Shape * ); // prototype
38
39 int main()
40 {
41     Circle circle;
42     Cylinder cylinder;
43     Shape *ptr = 0;
44
45     outputShapeArea( &circle ); // output circle's area

```

```

46     outputShapeArea( &cylinder ); // output cylinder's area
47     outputShapeArea( ptr );      // attempt to output area
48     return 0;
49 }
50
51 void outputShapeArea( const Shape *shapePtr )
52 {
53     const Circle *circlePtr;
54     const Cylinder *cylinderPtr;
55
56     // cast Shape * to a Cylinder *
57     cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );

```

Fig. 21.7 Demonstrating `dynamic_cast` (part 2 of 3).

```

58
59     if ( cylinderPtr != 0 ) // if true, invoke area()
60         cout << "Cylinder's area: " << cylinderPtr->area();
61     else { // shapePtr does not refer to a cylinder
62
63         // cast shapePtr to a Circle *
64         circlePtr = dynamic_cast< const Circle * >( shapePtr );
65
66         if ( circlePtr != 0 ) // if true, invoke area()
67             cout << "Circle's area: " << circlePtr->area();
68         else
69             cout << "Neither a Circle nor a Cylinder.";
70     }
71
72     cout << endl;
73 }

```

```

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
<i>Logical operator keywords</i>		
&&	and	logical AND
	or	logical OR
!	not	logical NOT
<i>Inequality operator keyword</i>		
!=	not_eq	inequality
<i>Bitwise operator keywords</i>		
&	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
~	<code>compl</code>	bitwise complement
<i>Bitwise assignment operator keywords</i>		
<code>&=</code>	<code>and_eq</code>	bitwise AND assignment
<code> =</code>	<code>or_eq</code>	bitwise inclusive OR assignment
<code>^=</code>	<code>xor_eq</code>	bitwise exclusive OR assignment

Fig. 21.8 Operator keywords as alternatives to operator symbols.

```

1 // Fig. 21.9: fig21_09.cpp
2 // Demonstrating operator keywords.
3 #include <iostream>
4 #include <iomanip>
5 #include <iso646.h>
6 using namespace std;
7
8 int main()
9 {
10     int a = 8, b = 22;
11
12     cout << boolalpha
13         << " a and b: " << ( a and b )
14         << "\n a or b: " << ( a or b )
15         << "\n not a: " << ( not a )
16         << "\na not_eq b: " << ( a not_eq b )
17         << "\na bitand b: " << ( a bitand b )
18         << "\na bit_or b: " << ( a bitor b )
19         << "\n a xor b: " << ( a xor b )
20         << "\n compl a: " << ( compl a )
21         << "\na and_eq b: " << ( a and_eq b )
22         << "\n a or_eq b: " << ( a or_eq b )
23         << "\na xor_eq b: " << ( a xor_eq b ) << endl;
24
25     return 0;
26 }
```

```

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.

```

1 // Fig 21.10: array2.h
2 // Simple class Array (for integers)
3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5
6 #include <iostream.h>
7
8 class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10 public:
11     Array( int = 10 ); // default/conversion constructor
12     ~Array(); // destructor
13 private:
14     int size; // size of the array
15     int *ptr; // pointer to first element of array
16 };
17
18 #endif

```

Fig. 21.10 Single-argument constructors and implicit conversions (part 1 of 4).

```

19 // Fig 21.10: array2.cpp
20 // Member function definitions for class Array
21 #include <assert.h>
22 #include "array2.h"
23
24 // Default constructor for class Array (default size 10)
25 Array::Array( int arraySize )
26 {
27     size = ( arraySize > 0 ? arraySize : 10 );
28     cout << "Array constructor called for "
29          << size << " elements\n";
30
31     ptr = new int[ size ]; // create space for array
32     assert( ptr != 0 ); // terminate if memory not allocated
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = 0; // initialize array
36 }
37

```

Fig. 21.10 Single-argument constructors and implicit conversions (part 2 of 4).

```

38 // Destructor for class Array
39 Array::~~Array() { delete [] ptr; }
40
41 // Overloaded output operator for class Array
42 ostream &operator<<( ostream &output, const Array &a )
43 {
44     int i;
45
46     for ( i = 0; i < a.size; i++ )
47         output << a.ptr[ i ] << ' ';
48
49     return output; // enables cout << x << y;
50 }

```

Fig. 21.10 Single-argument constructors and implicit conversions (part 3 of 4).

```

51 // Fig 21.10: fig21_10.cpp
52 // Driver for simple class Array
53 #include <iostream.h>
54 #include "array2.h"
55
56 void outputArray( const Array & );
57
58 int main()
59 {
60     Array integers1( 7 );
61
62     outputArray( integers1 ); // output Array integers1
63
64     outputArray( 15 ); // convert 15 to an Array and output
65
66     return 0;
67 }
68
69 void outputArray( const Array &arrayToOutput )
70 {
71     cout << "The array received contains:\n"
72          << arrayToOutput << "\n\n";
73 }

```

```

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).

```

1 // Fig. 21.11: array3.h
2 // Simple class Array (for integers)
3 #ifndef ARRAY1_H
4 #define ARRAY1_H
5
6 #include <iostream.h>
7
8 class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10 public:
11     explicit Array( int = 10 ); // default constructor
12     ~Array(); // destructor
13 private:
14     int size; // size of the array
15     int *ptr; // pointer to first element of array
16 };
17
18 #endif

```

Fig. 21.11 Demonstrating an **explicit** constructor (part 1 of 4).

```

19 // Fig. 21.11: array3.cpp
20 // Member function definitions for class Array
21 #include <assert.h>
22 #include "array3.h"

```

```

23
24 // Default constructor for class Array (default size 10)
25 Array::Array( int arraySize )
26 {
27     size = ( arraySize > 0 ? arraySize : 10 );
28     cout << "Array constructor called for "
29         << size << " elements\n";
30
31     ptr = new int[ size ]; // create space for array
32     assert( ptr != 0 );    // terminate if memory not allocated
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = 0;      // initialize array
36 }
37
38 // Destructor for class Array
39 Array::~Array() { delete [] ptr; }
40
41 // Overloaded output operator for class Array
42 ostream &operator<<( ostream &output, const Array &a )
43 {
44     int i;
45
46     for ( i = 0; i < a.size; i++ )
47         output << a.ptr[ i ] << ' ' ;
48
49     return output;    // enables cout << x << y;
50 }

```

Fig. 21.11 Demonstrating an **explicit** constructor (part 2 of 4).

```

51 // Fig. 21.11: fig21_11.cpp
52 // Driver for simple class Array
53 #include <iostream.h>
54 #include "array3.h"
55
56 void outputArray( const Array & );
57
58 int main()
59 {
60     Array integers1( 7 );
61
62     outputArray( integers1 );    // output Array integers1
63
64     outputArray( 15 );    // convert 15 to an Array and output
65 }

```

Fig. 21.11 Demonstrating an **explicit** constructor (part 3 of 4).

```

66     outputArray( Array( 15 ) ); // really want to do this!
67
68     return 0;
69 }
70
71 void outputArray( const Array &arrayToOutput )
72 {
73     cout << "The array received contains:\n"
74         << arrayToOutput << "\n\n";
75 }

```

```

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).

```

1 // Fig. 21.12: fig21_12.cpp
2 // Demonstrating storage class specifier mutable.
3 #include <iostream.h>
4
5 class TestMutable {
6 public:
7     TestMutable( int v = 0 ) { value = v; }
8     void modifyValue() const { value++; }
9     int getValue() const { return value; }
10 private:
11     mutable int value;
12 };
13
14 int main()
15 {
16     const TestMutable t( 99 );
17
18     cout << "Initial value: " << t.getValue();
19
20     t.modifyValue(); // modifies mutable member
21     cout << "\nModified value: " << t.getValue() << endl;
22
23     return 0;
24 }

```

```

Initial value: 99
Modified value: 100

```

Fig. 21.12 Demonstrating a **mutable** data member.

```

1 // Fig. 21.13: fig21_13.cpp
2 // Demonstrating operators .* and ->*
3 #include <iostream.h>
4
5 class Test {
6 public:
7     void function() { cout << "function\n"; }
8     int value;
9 };
10
11 void arrowStar( Test * );
12 void dotStar( Test * );
13
14 int main()

```



```

15 {
16     Test t;
17
18     t.value = 8;
19     arrowStar( &t );
20     dotStar( &t );
21     return 0;
22 }
23

```

Fig. 21.13 Demonstrating the `.*` and `->*` operators (part 1 of 2).

```

24 void arrowStar( Test *tPtr )
25 {
26     void ( Test::*memPtr )() = &Test::function;
27     ( tPtr->*memPtr )(); // invoke function indirectly
28 }
29
30 void dotStar( Test *tPtr )
31 {
32     int Test::*vPtr = &Test::value;
33     cout << ( *tPtr ).*vPtr << endl; // access value
34 }

```

```

function
8

```

Fig. 21.13 Demonstrating the `.*` and `->*` operators (part 2 of 2).

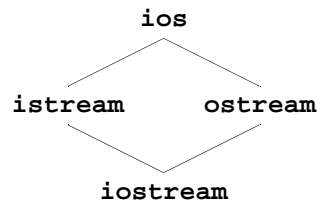


Fig. 21.14 Multiple inheritance to form class `iostream`.

```

1 // Fig. 21.15: fig21_15.cpp
2 // Attempting to polymorphically call a function
3 // multiply inherited from two base classes.
4 #include <iostream.h>
5
6 class Base {
7 public:
8     virtual void print() const = 0; // pure virtual
9 };
10
11 class DerivedOne : public Base {
12 public:
13     // override print function
14     void print() const { cout << "DerivedOne\n"; }
15 };
16
17 class DerivedTwo : public Base {
18 public:

```

```

19     // override print function
20     void print() const { cout << "DerivedTwo\n"; }
21 };
22
23 class Multiple : public DerivedOne, public DerivedTwo {
24 public:
25     // qualify which version of function print
26     void print() const { DerivedTwo::print(); }
27 };
28
29 int main()
30 {
31     Multiple both;    // instantiate Multiple object
32     DerivedOne one;  // instantiate DerivedOne object
33     DerivedTwo two;  // instantiate DerivedTwo object

```

Fig. 21.15 Attempting to call a multiply inherited function polymorphically (part 1 of 2).

```

34
35     Base *array[ 3 ];
36     array[ 0 ] = &both;    // ERROR--ambiguous
37     array[ 1 ] = &one;
38     array[ 2 ] = &two;
39
40     // polymorphically invoke print
41     for ( int k = 0; k < 3; k++ )
42         array[ k ] -> print();
43
44     return 0;
45 }

```

```

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).

```

1 // Fig. 21.16: fig21_16.cpp
2 // Using virtual base classes.
3 #include <iostream.h>
4
5 class Base {
6 public:
7     // implicit default constructor
8
9     virtual void print() const = 0; // pure virtual
10 };
11
12 class DerivedOne : virtual public Base {
13 public:
14     // implicit default constructor calls
15     // Base default constructor
16
17     // override print function
18     void print() const { cout << "DerivedOne\n"; }
19 };
20

```

```

21 class DerivedTwo : virtual public Base {
22 public:
23     // implicit default constructor calls
24     // Base default constructor
25
26     // override print function
27     void print() const { cout << "DerivedTwo\n"; }
28 };
29
30 class Multiple : public DerivedOne, public DerivedTwo {
31 public:
32     // implicit default constructor calls
33     // DerivedOne and DerivedTwo default constructors
34
35     // qualify which version of function print
36     void print() const { DerivedTwo::print(); }
37 };
38
39 int main()
40 {
41     Multiple both;    // instantiate Multiple object
42     DerivedOne one;  // instantiate DerivedOne object
43     DerivedTwo two;  // instantiate DerivedTwo object
44
45     Base *array[ 3 ];
46     array[ 0 ] = &both;
47     array[ 1 ] = &one;
48     array[ 2 ] = &two;
49

```

Fig. 21.16 Using **virtual** base classes (part 1 of 2).

```

50     // polymorphically invoke print
51     for ( int k = 0; k < 3; k++ )
52         array[ k ] -> print();
53
54     return 0;
55 }

```

```

DerivedTwo
DerivedOne
DerivedTwo

```

Fig. 21.16 Using **virtual** base classes (part 2 of 2).