

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**.

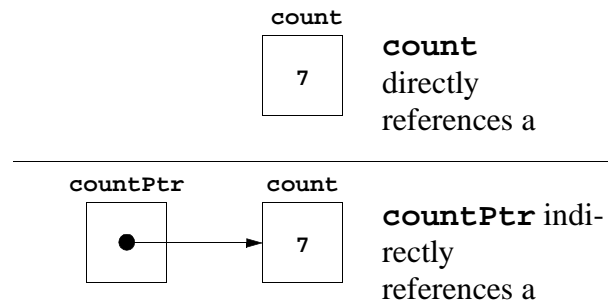


Fig. 5.1 Directly and indirectly referencing a variable.

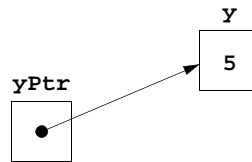


Fig. 5.2 Graphical representation of a pointer pointing to an integer variable in memory.



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

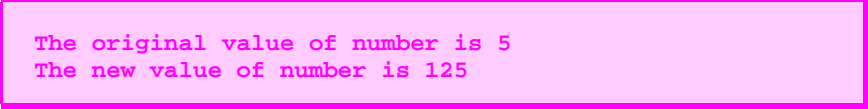
```

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;
```

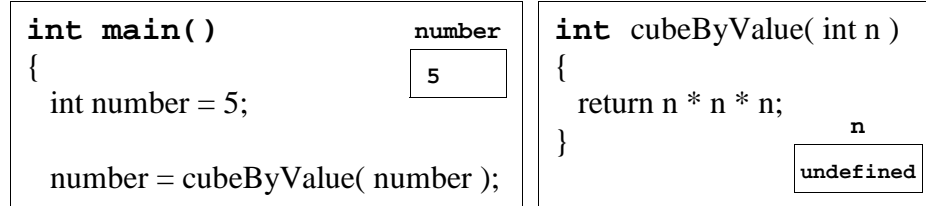
Fig. 5.7 Cube a variable using call-by-reference with a pointer argument (part 1 of 2).

```
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 }
```

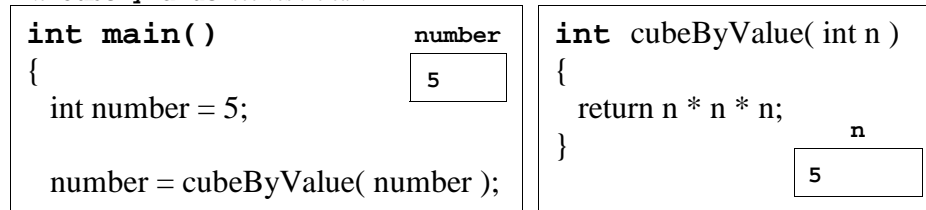
The original value of number is 5
The new value of number is 125

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

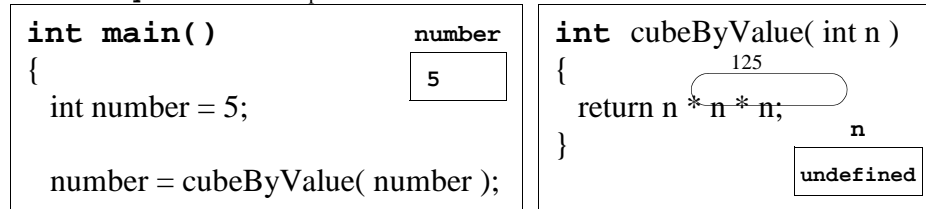
Before **main** calls **cubeByValue**:



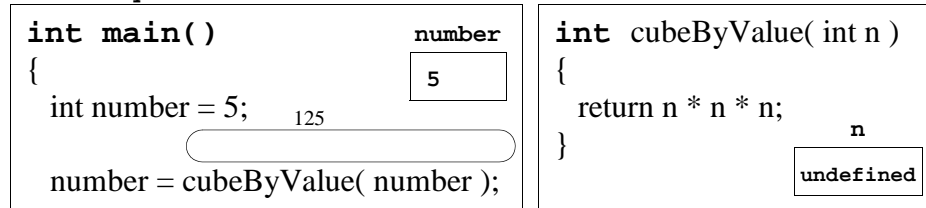
After **cubeByValue** receives the call:



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



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



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

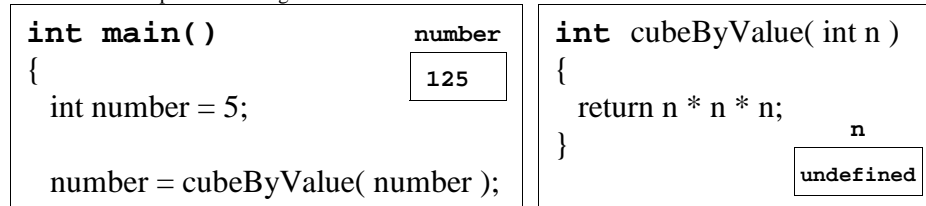
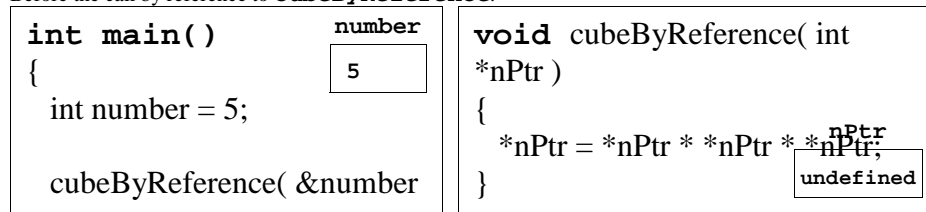
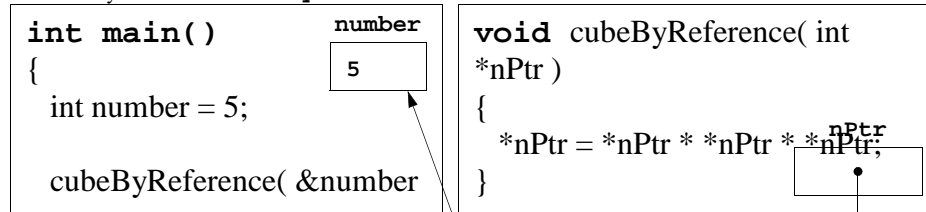


Fig. 5.8 Analysis of a typical call-by-value.

Before the call by reference to `cubeByReference`:



After call by reference to `cubeByReference` and before `*nPTr` is cubed:



After `*nPTr` is cubed:

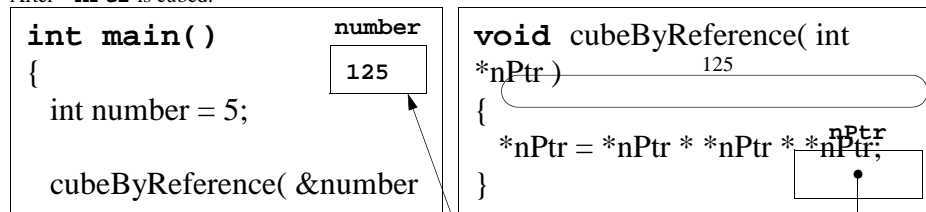


Fig. 5.9 Analysis of a typical call-by-reference with a pointer argument.

```

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;

```

<p>Data items in original order</p> <p>2 6 4 8 10 12 89 68 45 37</p> <p>Data items in ascending order</p> <p>2 4 6 8 10 12 37 45 68 89</p>
--

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).

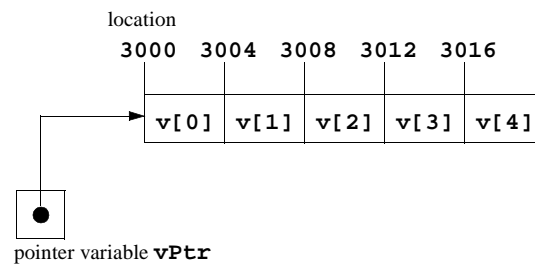


Fig. 5.18 The array **v** and a pointer variable **vPtr** that points to **v**.

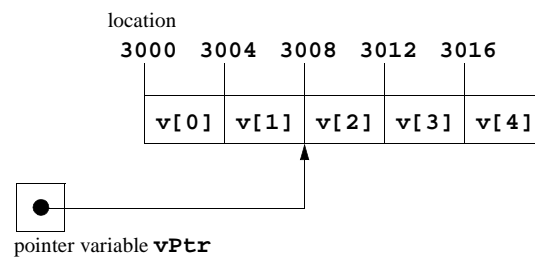


Fig. 5.19 The pointer **vPtr** after pointer arithmetic.

```

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).

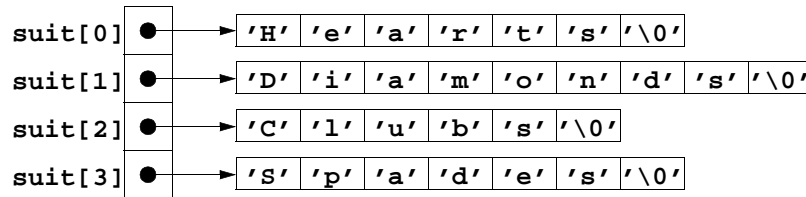


Fig. 5.22 A graphical representation of the `suit` array.

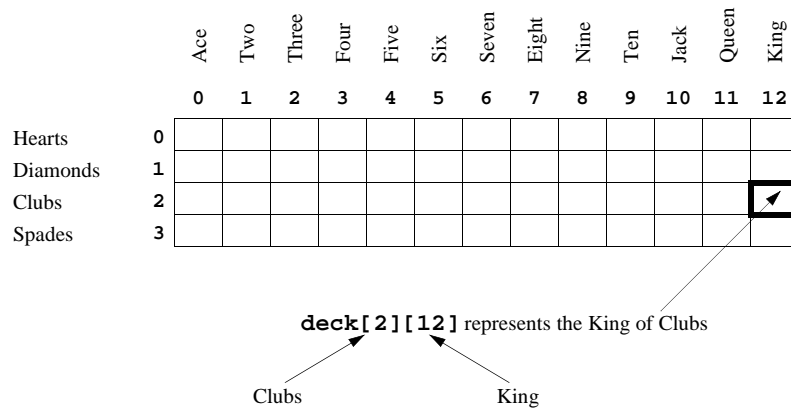


Fig. 5.23 Double-subscripted array representation of a deck of cards.

```

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         for ( int row = 0; row <= 3; row++ )
48             for ( int column = 0; column <= 12; column++ )
49                 if ( wDeck[ row ][ column ] == card )
50                     cout << setw( 5 ) << setiosflags( ios::right )
51                         << wFace[ column ] << " of "
52                         << setw( 8 ) << setiosflags( ios::left )
53                         << wSuit[ row ]
54                         << ( card % 2 == 0 ? '\n' : '\t' );
55 }
56
57
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
48     for ( int count = 0; count < size - 1; count++ )
49
50         if ( (*compare)( work[ count ], work[ count + 1 ] ) )
51             swap( &work[ count ], &work[ count + 1 ] );
52 }
53
54 void swap( int *element1Ptr, int *element2Ptr )
55 {
56     int temp;
57
58     temp = *element1Ptr;
59     *element1Ptr = *element2Ptr;
60     *element2Ptr = temp;
61 }
62
63 int ascending( int a, int b )
64 {
65     return b < a;    // swap if b is less than a
66 }
67
68 int descending( int a, int b )
69 {
70     return b > a;    // swap if b is greater than a
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         << "\nThe length of \"" << string2
15         << "\" is " << strlen( string2 )
16         << "\nThe length of \"" << string3
17         << "\" is " << strlen( string3 ) << endl;
18
19    return 0;
20 }

```

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

Fig. 5.34 Using `strlen`.