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`**, **`iscntrl`**, **`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 **`strtol`**.
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`**.

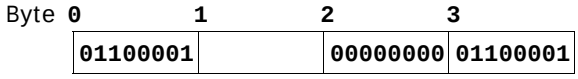

Fig. 16.1 A possible storage alignment for a variable of type **`Example`** showing an undefined area in memory.

```
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
32    cout << endl;
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 (**&**).

```
// Fig. 16.7: fig16_07.cpp
// Using the bitwise AND, bitwise inclusive OR, bitwise
// exclusive OR, and bitwise complement operators.
#include <iostream.h>
#include <iomanip.h>

void displayBits( unsigned );

int main()
{
   unsigned number1, number2, mask, setBits;

   number1 = 65535;
   mask = 1;
```

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

```
   cout << "The result of combining the following\n";
   displayBits( number1 );
   displayBits( mask );
   cout << "using the bitwise AND operator & is\n";
   displayBits( number1 & mask );

   number1 = 15;
   setBits = 241;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( setBits );
   cout << "using the bitwise inclusive OR operator | is\n";
   displayBits( number1 | setBits );

   number1 = 139;
   number2 = 199;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( number2 );
   cout << "using the bitwise exclusive OR operator ^ is\n";
   displayBits( number1 ^ number2 );

```

```
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
60     cout << endl;
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
`int isdigit( int c )`	Returns **`true`** if **`c`** is a digit, and **`false`** otherwise.
`int isalpha( int c )`	Returns **`true`** if **`c`** is a letter, and **`false`** otherwise.
`int isalnum( int c )`	Returns **`true`** if **`c`** is a digit or a letter, and **`false`** otherwise.
`int isxdigit( int c )`	Returns **`true`** if **`c`** is a hexadecimal digit character, and **`false`** otherwise. (See Appendix C, "Number Systems," for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
`int islower( int c )`	Returns **`true`** if **`c`** is a lowercase letter, and **`false`** otherwise.
`int isupper( int c )`	Returns **`true`** if **`c`** is an uppercase letter; **`false`** otherwise.
`int tolower( int c )`	If **`c`** is an uppercase letter, **`tolower`** returns **`c`** as a lowercase letter. Otherwise, **`tolower`** returns the argument unchanged.

Fig. 16.16 Summary of the character handling library functions (part 1 of 2).

Prototype	Description
`int toupper( int c )`	If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c )`	Returns `true` if `c` is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and `false` otherwise
`int iscntrl( int c )`	Returns `true` if `c` is a control character, and `false` otherwise.
`int ispunct( int c )`	Returns `true` if `c` is a printing character other than a space, a digit, or a letter, and `false` otherwise.
`int isprint( int c )`	Returns `true` value if `c` is a printing character including space (`' '`), and `false` otherwise.
`int isgraph( int c )`	Returns `true` if `c` is a printing character other than space (`' '`), and `false` otherwise.

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

```
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"
```

```
         << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
         << " hexadecimal digit\n"
         << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
         << " hexadecimal digit\n"
         << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
         << " hexadecimal digit" << endl;
   return 0;
}
```

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

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

```
// Fig. 16.18: fig16_18.cpp
// Using functions islower, isupper, tolower, toupper
#include <iostream.h>
#include <ctype.h>

int main()
{
   cout << "According to islower:\n"
        << ( islower( 'p' ) ? "p is a" : "p is not a" )
        << " lowercase letter\n"
        << ( islower( 'P' ) ? "P is a" : "P is not a" )
        << " lowercase letter\n"
        << ( islower( '5' ) ? "5 is a" : "5 is not a" )
        << " lowercase letter\n"
        << ( islower( '!' ) ? "! is a" : "! is not a" )
        << " lowercase letter\n";
   cout << "\nAccording to isupper:\n"
        << ( isupper( 'D' ) ? "D is an" : "D is not an" )
        << " uppercase letter\n"
        << ( isupper( 'd' ) ? "d is an" : "d is not an" )
        << " uppercase letter\n"
        << ( isupper( '8' ) ? "8 is an" : "8 is not an" )
        << " uppercase letter\n"
        << ( isupper('$') ? "$ is an" : "$ is not an" )
        << " uppercase letter\n";
   cout << "\nu converted to uppercase is "
        << ( char ) toupper( 'u' )
        << "\n7 converted to uppercase is "
```

```
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**, **iscntrl**, **ispunct**, **isprint**, and **isgraph** (part 2 of 3).

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

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

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

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

```
// Fig. 16.21: fig16_21.cpp
// Using atof
#include <iostream.h>
#include <stdlib.h>

int main()
{
   double d = atof( "99.0" );

   cout << "The string \"99.0\" converted to double is "
        << d << "\nThe converted value divided by 2 is "
        << d / 2.0 << endl;
   return 0;
}
```

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

Fig. 16.21 Using `atof`.

```
// Fig. 16.22: fig16_22.cpp
// Using atoi
#include <iostream.h>
#include <stdlib.h>

int main()
{
   int i = atoi( "2593" );

   cout << "The string \"2593\" converted to int is " << i
        << "\nThe converted value minus 593 is " << i - 593
        << endl;
   return 0;
}
```

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

Fig. 16.22 Using `atoi`.

```
// Fig. 16.23: fig16_23.cpp
// Using atol
#include <iostream.h>
#include <stdlib.h>

int main()
{
   long l = atol( "1000000" );

   cout << "The string \"1000000\" converted to long is " << l
        << "\nThe converted value divided by 2 is " << l / 2
        << endl;
   return 0;
}
```

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

Fig. 16.23 Using `atol`.

```
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         << "\"\nThe converted value is " << x
14         << "\nThe remainder of the original string is \""
15         << remainderPtr
16         << "\"\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
`char *strchr( const char *s, int c )`	Locates the first occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in `s` is returned. Otherwise, a `NULL` pointer is returned.
`size_t strcspn( const char *s1, const char *s2 )`	Determines and returns the length of the initial segment of string `s1` consisting of characters not contained in string `s2`.
`size_t strspn( const char *s1, const char *s2 )`	Determines and returns the length of the initial segment of string `s1` consisting only of characters contained in string `s2`.
`char *strpbrk( const char *s1, const char *s2 )`	

Fig. 16.27 Search functions of the string handling library.

Prototype	Description
	Locates the first occurrence in string **s1** of any character in string **s2**. If a character from string **s2** is found, a pointer to the character in string **s1** is returned. Otherwise a **NULL** pointer is returned.
`char *strrchr( const char *s, int c )`	
	Locates the last occurrence of **c** in string **s**. If **c** is found, a pointer to **c** in string **s** is returned. Otherwise, a **NULL** pointer is returned.
`char *strstr( const char *s1, const char *s2 )`	
	Locates the first occurrence in string **s1** of string **s2**. If the string is found, a pointer to the string in **s1** is returned. Otherwise, a **NULL** pointer is returned.

Fig. 16.27 Search functions of the string handling library.

```
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          << "\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 \"" << string2 << "\"\n'"
12         << *strpbrk( string1, string2 ) << '\''
13         << " is the first character to appear in\n\""
14         << string1 << '\"' << endl;
15    return 0;
16 }
```

```
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 strrchr
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: \"" << strrchr( string1, c ) << '\"' << endl;
14    return 0;
15 }
```

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

Fig. 16.31 Using `strrchr`.

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

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

```
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 << '\"' << 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[] = "ABCDEFG", 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 = ABCDEFG
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`.