EE1003
Introduction to Computer I

# Chapter 22
# Bits, Characters, C Strings and structs

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw



## Learning Objectives

In this chapter you'll learn:

- To create and use `structs`.
- To pass `structs` by value and by reference.
- To use `typedef` to create aliases for previously defined data types and `structs`.
- To manipulate data with the bitwise operators and to create bit fields for storing data compactly.
- To use the functions of the character-handling library `<cctype>`.
- To use the string-conversion functions of the general-utilities library `<cstdlib>`.
- To use the string-processing functions of the string-handling library `<cstring>`.

# Outline





# 22.1 Introduction

- We now discuss structures and the manipulation of bits, characters and C-style strings.
- Many of the techniques we present here are included for the benefit of those who will work with legacy C and C++ code.
- We discuss how to declare, initialize and pass structures to functions.
- Then, we present a high-performance card shuffling and dealing simulation in which we use structure objects and C-style strings to represent the cards.

# 22.1 Introduction

- We discuss the bitwise operators that allow you to access and manipulate the individual bits in bytes of data.
- We also present bitfields—special structures that can be used to specify the exact number of bits a variable occupies in memory.
- These bit manipulation techniques are common in C and C++ programs that interact directly with hardware devices that have limited memory.



# 22.2 Structure Definitions

- Structures are aggregate data types—that is, they can be built using elements of several types including other `structs`.
- Example

```
struct card {
   char *face;
   char *suit;
};
```

  - `struct` introduces the definition for structure `card`
  - `card` is the structure tag and is used to declare variables of the structure type
  - `card` contains two members of type `char *`
    - These members are `face` and `suit`

# 22.2 Structure Definitions

- Another Example

```
struct employee{
    char firstName[20];
    char lastName[20];
    int age;
    char gender;
    double hourlySalary;
};
```

**Common Programming Error 22.1**

*Forgetting the semicolon that terminates a structure definition is a syntax error.*

- Members of the same structure must have unique names, but two different structures may contain members of the same name without conflict.
- Each structure definition must end with a semicolon.



# 22.2 Structure Definitions

- A structure definition does not reserve any space in memory; rather, it creates a new data type that is used to declare structure variables.
- Structure variables are declared like variables of other types.
- Variables of a given structure type can also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition.
- The structure name is optional.
- If a structure definition does not contain a structure name, variables of the structure type may be declared only between the closing right brace of the structure definition and the semicolon that terminates the structure definition.

# 22.2 Structure Definitions

**Software Engineering Observation 22.1**

*Provide a structure name when creating a structure type. The structure name is required for declaring new variables of the structure type later in the program, declaring parameters of the structure type and, if the structure is being used like a C++ class, specifying the name of the constructor and destructor.*



# 22.2 Structure Definitions

- `struct` information
  - A `struct` cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to define structure variables
- Structure variables
  - Defined like other variables:

```
struct card oneCard, deck[ 52 ], *cPtr;
```

  - Can be defined together with a structure definition:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

## 22.2 Structure Definitions

- Valid operations
  - Assigning a structure to a structure of the same type
  - Taking the address (&) of a structure
  - Accessing the members of a structure
  - Using the `sizeof` operator to determine the size of a structure



## 22.2 Structure Definitions

- Comparing structures is a syntax error
- Structure members are not necessarily stored in consecutive bytes of memory.
- Sometimes there are "holes" in a structure, because some computers store specific data types only on certain memory boundaries for performance reasons, such as half-word, word or double-word boundaries.
- A word is a standard memory unit used to store data in a computer—usually two bytes or four bytes and typically four bytes on today's popular 32-bit systems.

## 22.2 Structure Definitions

- Consider the following structure definition in which structure objects `sample1` and `sample2` of type `Example` are declared:

```
struct Example
{
   char c;
   int i;
} sample1, sample2;
```

- A computer with two-byte words might require that each of the members of `Example` be aligned on a word boundary (i.e., at the beginning of a word—this is machine dependent).



## 22.2 Structure Definitions

- Figure 22.1 shows a sample storage alignment for an object of type `Example` that has been assigned the character `'a'` and the integer `97` (the bit representations of the values are shown).
- If the members are stored beginning at word boundaries, there is a one-byte hole (byte `1` in the figure) in the storage for objects of type `Example`.
- The value in the one-byte hole is undefined.
- If the member values of `sample1` and `sample2` are in fact equal, the structure objects are not necessarily equal, because the undefined one-byte holes are not likely to contain identical values.

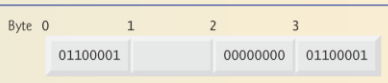



**Fig. 22.1** | Possible storage alignment for a variable of type `Example`, showing an undefined area in memory.

# 22.3 Initializing Structures

- Structures can be initialized using initializer lists, like arrays.
- If there are fewer initializers in the list than members in the structure, the remaining members are initialized to their default values.
- Structure variables declared outside a function definition (i.e., externally) are initialized to their default values if they're not explicitly initialized in the external declaration.
- Structure variables may also be set in assignment expressions by assigning a structure variable of the same type or by assigning values to the individual data members of the structure.



# 22.2 Structure Definitions

```
struct card {
  char *face;
  char *suit;
 };
```

- Initializer lists
  - Example:

```
struct card oneCard = { "Three", "Hearts" };
```

- Assignment statements
  - Example:

```
struct card threeHearts = oneCard;
```

  - Could also define and initialize `threeHearts` as follows:

```
struct card threeHearts;
threeHearts.face = "Three";
threeHearts.suit = "Hearts";
```

# 22.2 Structure Definitions

- Accessing structure members
  - Dot operator ( . ) used with structure variables

```
struct card myCard;
cout << myCard.suit;
```

  - Arrow operator (->) used with pointers to structure variables

```
struct card *myCardPtr = &myCard;
cout<< myCardPtr->suit;
```

  - `myCardPtr->suit` is equivalent to

```
( *myCardPtr ).suit
```



# 22.2 Structure Definitions

```
#include <iostream>

using namespace std;

struct card{
    char *face;
    char *suit;
};


int main()
{
    card a;
    card *aptr;

    a.face="Ace";
    a.suit="Spades";

    aptr=&a;

    cout<<a.face<<" of "<<a.suit<<endl;
    cout<<aptr->face<<" of "<<aptr->suit<<endl;
    cout<<(*aptr).face<<" of "<<(*aptr).suit<<endl;

    return 0;

}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

## 22.4 Using Structures with Functions

- There are two ways to pass the information in structures to functions.
- You can either pass the entire structure or pass the individual members of a structure.
- By default, structures are passed by value.
- Structures and their members can also be passed by reference by passing either references or pointers.
- To pass a structure by reference, pass the address of the structure object or a reference to the structure object.



## 22.4 Using Structures with Functions

- In Chapter 7, we stated that an array could be passed by value by using a structure.
- To pass an array by value, create a structure (or a class) with the array as a member, then pass an object of that structure (or class) type to a function by value.
- Because structure objects are passed by value, the array member, too, is passed by value.

**Performance Tip 22.1**

*Passing structures (and especially large structures) by reference is more efficient than passing them by value (which requires the entire structure to be copied).*

# 22.5 typedef

- Keyword typedef provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter, simpler or more readable type names.
- For example, the statement
  - `typedef Card *CardPtr;`

  defines the new type name `CardPtr` as a synonym for type `Card *`.
- Creating a new name with `typedef` does not create a new type; `typedef` simply creates a new type name that can then be used in the program as an alias for an existing type name.



# 22.5 typedef

**Good Programming Practice 22.1**

*Capitalize `typedef` names to emphasize that they are synonyms for other type names.*

**Portability Tip 22.2**

*Synonyms for built-in data types can be created with `typedef` to make programs more portable. For example, a program can use `typedef` to create alias `Integer` for four-byte integers. `Integer` can then be aliased to `int` on systems with four-byte integers and can be aliased to `long int` on systems with two-byte integers where `long int` values occupy four bytes. Then, you simply declare all four-byte integer variables to be of type `Integer`.*

# 22.6 Example: Card Shuffling and Dealing Simulation

- Pseudocode:
  - *Create an array of card structures*
  - *Put cards in the deck*
  - *Shuffle the deck*
  - *Deal the cards*



# 22.6 Example: Card Shuffling and Dealing Simulation

```
#include <iostream>
#include <time.h>

using namespace std;

struct card{
    const char *face;
    const char *suit;
};

typedef struct card Card;

void fillDeck(Card * const wDeck, const char *wFace[], const char *wSuit[]);
void shuffle(Card * const wDeck);
void deal(const Card * const wDeck);

int main()
{
    Card deck[52];
    const char *face[]={"Ace","Deuce","Three","Four","Five","Six","Seven","Eight","Nine","Ten","Jack","Queen","King"};
    const char *suit[]={"Hearts","Diamonds","Clubs","Spades"};

    srand(time(NULL));

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

    return 0;
}
```

# 22.6 Example: Card Shuffling and Dealing Simulation

```
32  void fillDeck(Card * const wDeck, const char *wFace[], const char *wSuit[]){
33      for(int i=0;i<=51;i++){
34          wDeck[i].face=wFace[i%13];
35          wDeck[i].suit=wSuit[i%13];
36      }
37
38  }
39  void shuffle(Card * const wDeck){
40      for(int i=0;i<=51;i++){
41          int j=rand()%52;
42          Card temp=wDeck[i];
43          wDeck[i]=wDeck[j];
44          wDeck[j]=temp;
45      }
46  }
47  void deal(const Card * const wDeck){
48      for(int i=0;i<=51;i++){
49          cout<<wDeck[i].face<<" of "<<wDeck[i].suit<<endl;
50      }
51  }
52
```



# 22.6 Example: Card Shuffling and Dealing Simulation

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

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

## Supplementary: Unions

- union
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share space
  - Conserves storage
- union definitions
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```



## Supplementary: Unions

- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->

# Supplementary: Unions

```
1   #include <iostream>
2
3   using namespace std;
4
5   union number{
6       int x;
7       double y;
8   };
9
10  int main()
11  {
12      number value;
13      value.x=100;
14
15      cout << "Put a value in the integer member" << endl;
16      cout << "and print both member."<<endl;
17      cout << "int:    "<<value.x<<endl;
18      cout << "double:    "<<value.y<<endl;
19
20      cout<<endl;
21
22      value.y=100.001;
23      cout << "Put a value in the floating member" << endl;
24      cout << "and print both member."<<endl;
25      cout << "int:    "<<value.x<<endl;
26      cout << "double:    "<<value.y<<endl;
27
28      return 0;
29  }
30
```

```
Put a value in the integer member
and print both member.
int:    100
double:    4.94066e-322

Put a value in the floating member
and print both member.
int:    1649267442
double:    100.001
```



# 22.7 Bitwise Operators

- C++ provides extensive bit-manipulation capabilities for getting down to the so-called “bits-and-bytes” level.
- Operating systems, test-equipment software, networking software and many other kinds of software require that you communicate “directly with the hardware.”
- We introduce each of C++’s many bitwise operators, and we discuss how to save memory by using bit fields.

## 22.7 Bitwise Operators

- All data is represented internally by computers as sequences of bits.
- Each bit can assume the value `0` or the value `1`.
- On most systems, a sequence of 8 bits forms a byte—the standard storage unit for a variable of type `char`.
- Other data types are stored in larger numbers of bytes.
- Bitwise operators are used to manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both `signed` and `unsigned`).
- Unsigned integers are normally used with the bitwise operators.



## 22.7 Bitwise Operators

- The bitwise operator discussions in this section show the binary representations of the integer operands.
  - For a detailed explanation of the binary (also called base-2) number system, see Appendix D, Number Systems.
- Because of the machine-dependent nature of bitwise manipulations, some of these programs might not work on your system without modification.
- The bitwise operators are: bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive OR (^), left shift (<<), right shift (>>) and bitwise complement (~)—also known as the one's complement.

## 22.7 Bitwise Operators

- The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit.
- The bitwise AND operator sets each bit in the result to 1 if the corresponding bit in both operands is 1.
- The bitwise inclusive-OR operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1.
- The bitwise exclusive-OR operator sets each bit in the result to 1 if the corresponding bit in either operand—but not both—is 1.



## 22.7 Bitwise Operators

- The left-shift operator shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- The right-shift operator shifts the bits in its left operand to the right by the number of bits specified in its right operand.
- The bitwise complement operator sets all 0 bits in its operand to 1 in the result and sets all 1 bits in its operand to 0 in the result.
- The bitwise operators are summarized in Fig. 22.5.

# 22.7 Bitwise Operators

| 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 one or both 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. |
| ~ | bitwise complement | All 0 bits are set to 1 and all 1 bits are set to 0. |

**Fig. 22.5** | Bitwise operators.



# 22.7 Bitwise Operators

- When using the bitwise operators, it's useful to illustrate their precise effects by printing values in their binary representation.
- The program of Fig. 22.6 prints an `unsigned` integer in its binary representation in groups of eight bits each.

# 22.7 Bitwise Operators

```
1  // Fig. 22.6: fig22_06.cpp
2  // Printing an unsigned integer in bits.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  void displayBits( unsigned ); // prototype
8
9  int main()
10 {
11    unsigned inputValue; // integral value to print in binary
12
13    cout << "Enter an unsigned integer: ";
14    cin >> inputValue;
15    displayBits( inputValue );
16 } // end main
17
```

**Fig. 22.6** | Printing an unsigned integer in bits. (Part 1 of 3.)



# 22.7 Bitwise Operators

```
18 // display bits of an unsigned integer value
19 void displayBits( unsigned value )
20 {
21    const int SHIFT = 8 * sizeof( unsigned ) - 1;
22    const unsigned MASK = 1 << SHIFT;
23
24    cout << setw( 10 ) << value << " = ";
25
26    // display bits
27    for ( unsigned i = 1; i <= SHIFT + 1; i++ )
28    {
29       cout << ( value & MASK ? '1' : '0' );
30       value <<= 1; // shift value left by 1
31
32       if ( i % 8 == 0 ) // output a space after 8 bits
33          cout << ' ';
34    } // end for
35
36    cout << endl;
37 } // end function displayBits
```

**Fig. 22.6** | Printing an unsigned integer in bits. (Part 2 of 3.)

# 22.7 Bitwise Operators

```
Enter an unsigned integer: 65000
     65000 = 00000000 00000000 11111101 11101000
```

```
Enter an unsigned integer: 29
        29 = 00000000 00000000 00000000 00011101
```

**Fig. 22.6** | Printing an unsigned integer in bits. (Part 3 of 3.)



# 22.7 Bitwise Operators

- Function `displayBits` (lines 19–37) uses the bitwise AND operator to combine variable `value` with constant `MASK`.
- Often, the bitwise AND operator is used with an operand called a mask—an integer value with specific bits set to `1`.
- Masks are used to hide some bits in a value while selecting other bits.
- In `displayBits`, line 22 assigns constant `MASK` the value `1 << SHIFT`.

## 22.7 Bitwise Operators

- The value of constant `SHIFT` was calculated in line 21 with the expression
  - `8 * sizeof( unsigned ) - 1`
- which multiplies the number of bytes an `unsigned` object requires in memory by `8` (the number of bits in a byte) to get the total number of bits required to store an `unsigned` object, then subtracts 1.
- The bit representation of `1 << SHIFT` on a computer that represents `unsigned` objects in four bytes of memory is
  - `10000000 00000000 00000000 00000000`
- The left-shift operator shifts the value `1` from the low-order (rightmost) bit to the high-order (leftmost) bit in `MASK`, and fills in `0` bits from the right.



## 22.7 Bitwise Operators

- Line 29 prints a `1` or a `0` for the current leftmost bit of variable `value`.
- Assume that variable `value` contains `65000` (`00000000 00000000 11111101 11101000`).
- When `value` and `MASK` are combined using `&`, all the bits except the high-order bit in variable `value` are "masked off" (hidden), because any bit "ANDed" with `0` yields `0`.
- If the leftmost bit is `1`, `value & MASK` evaluates to

```
00000000 00000000 11111101 11101000     (value)
10000000 00000000 00000000 00000000     (MASK)
---------------------------------
00000000 00000000 00000000 00000000     (value & MASK)
```

- which is interpreted as `false`, and `0` is printed.
- Then line 30 shifts variable `value` left by one bit with the expression `value <<= 1` (i.e., `value = value << 1`).
- These steps are repeated for each bit variable `value`.

# 22.7 Bitwise Operators

- Eventually, a bit with a value of 1 is shifted into the leftmost bit position, and the bit manipulation is as follows:

```
11111101 11101000 00000000 00000000    (value)
10000000 00000000 00000000 00000000    (MASK)
--------------------------------
10000000 00000000 00000000 00000000    (value & MASK)
```

- Because both left bits are 1s, the expression's result is nonzero (true) and 1 is printed.
- Figure 22.7 summarizes the results of combining two bits with the bitwise AND operator.



# 22.7 Bitwise Operators

**Common Programming Error 22.3**

*Using the logical AND operator (&&) for the bitwise AND operator (&) and vice versa is a logic error.*

| Bit 1 | Bit 2 | Bit 1 & Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

**Fig. 22.7** | Results of combining two bits with the bitwise AND operator (&).

# 22.7 Bitwise Operators

- The program of Fig. 22.8 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator.
- Function `displayBits` (lines 53–71) prints the `unsigned` integer values.



# 22.7 Bitwise Operators

```
1  // Fig. 22.8: fig22_08.cpp
2  // Bitwise AND, inclusive OR,
3  // exclusive OR and complement operators.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  void displayBits( unsigned ); // prototype
9
10 int main()
11 {
12    unsigned number1;
13    unsigned number2;
14    unsigned mask;
15    unsigned setBits;
16
17    // demonstrate bitwise &
18    number1 = 2179876355;
19    mask = 1;
20    cout << "The result of combining the following\n";
21    displayBits( number1 );
22    displayBits( mask );
```

**Fig. 22.8** | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 1 of 5.)

# 22.7 Bitwise Operators

```
23     cout << "using the bitwise AND operator & is\n";
24     displayBits( number1 & mask );
25
26     // demonstrate bitwise |
27     number1 = 15;
28     setBits = 241;
29     cout << "\nThe result of combining the following\n";
30     displayBits( number1 );
31     displayBits( setBits );
32     cout << "using the bitwise inclusive OR operator | is\n";
33     displayBits( number1 | setBits );
34
35     // demonstrate bitwise exclusive OR
36     number1 = 139;
37     number2 = 199;
38     cout << "\nThe result of combining the following\n";
39     displayBits( number1 );
40     displayBits( number2 );
41     cout << "using the bitwise exclusive OR operator ^ is\n";
42     displayBits( number1 ^ number2 );
43
```

**Fig. 22.8** | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 2 of 5.)



# 22.7 Bitwise Operators

```
44     // demonstrate bitwise complement
45     number1 = 21845;
46     cout << "\nThe one's complement of\n";
47     displayBits( number1 );
48     cout << "is" << endl;
49     displayBits( ~number1 );
50  } // end main
51
```

**Fig. 22.8** | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 3 of 5.)

# 22.7 Bitwise Operators

```
52  // display bits of an unsigned integer value
53  void displayBits( unsigned value )
54  {
55     const int SHIFT = 8 * sizeof( unsigned ) - 1;
56     const unsigned MASK = 1 << SHIFT;
57
58     cout << setw( 10 ) << value << " = ";
59
60     // display bits
61     for ( unsigned i = 1; i <= SHIFT + 1; i++ )
62     {
63        cout << ( value & MASK ? '1' : '0' );
64        value <<= 1; // shift value left by 1
65
66        if ( i % 8 == 0 ) // output a space after 8 bits
67           cout << ' ';
68     } // end for
69
70     cout << endl;
71  } // end function displayBits
```

**Fig. 22.8** | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 4 of 5.)



# 22.7 Bitwise Operators

```
The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100

The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

**Fig. 22.8** | Bitwise AND, inclusive-OR, exclusive-OR and complement operators. (Part 5 of 5.)

## 22.7 Bitwise Operators

- In Fig. 22.8, line 18 assigns `2179876355` (`10000001 11101110 01000110 00000011`) to variable `number1`, and line 19 assigns `1` (`00000000 00000000 00000000 00000001`) to variable `mask`.
- When `mask` and `number1` are combined using the bitwise AND operator (`&`) in the expression `number1 & mask` (line 24), the result is `00000000 00000000 00000000 00000001`.
- All the bits except the low-order bit in variable `number1` are "masked off" (hidden) by "ANDing" with constant `MASK`.



## 22.7 Bitwise Operators

- The bitwise inclusive-OR operator is used to set specific bits to 1 in an operand.
- In Fig. 22.8, line 27 assigns `15` (`00000000 00000000 00000000 00001111`) to variable `number1`, and line 28 assigns `241` (`00000000 00000000 00000000 11110001`) to variable `setBits`.
- When `number1` and `setBits` are combined using the bitwise OR operator in the expression `number1 | setBits` (line 33), the result is `255` (`00000000 00000000 00000000 11111111`).
- Figure 22.9 summarizes the results of combining two bits with the bitwise inclusive-OR operator.

# 22.7 Bitwise Operators

**Common Programming Error 22.4**

*Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is a logic error.*

| Bit 1 | Bit 2 | Bit 1 \| Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

**Fig. 22.9** | Combining two bits with the bitwise inclusive-OR operator (|).



# 22.7 Bitwise Operators

- The bitwise exclusive OR operator (^) sets each bit in the result to 1 if *exactly one of the corresponding bits in its two operands is 1*.
- In Fig. 22.8, lines 36–37 assign variables `number1` and `number2` the values `139` (`00000000 00000000 00000000 10001011`) and `199` (`00000000 00000000 00000000 11000111`), respectively.
- When these variables are combined with the exclusive-OR operator in the expression `number1 ^ number2` (line 42), the result is `00000000 00000000 00000000 01001100`.
- Figure 22.10 summarizes the results of combining two bits with the bitwise exclusive-OR operator.

# 22.7 Bitwise Operators

| Bit 1 | Bit 2 | Bit 1 ^ Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

**Fig. 22.10** | Combining two bits with the bitwise exclusive-OR operator (^).



# 22.7 Bitwise Operators

- The bitwise complement operator (~) sets all `1` bits in its operand to `0` in the result and sets all `0` bits to `1` in the result—otherwise referred to as "taking the one's complement of the value."
- In Fig. 22.8, line 45 assigns variable `number1` the value `21845` (00000000 00000000 01010101 01010101).
- When the expression `~number1` evaluates, the result is (11111111 11111111 10101010 10101010).
- Figure 22.11 demonstrates the left-shift operator (<<) and the right-shift operator (>>).
- Function `displayBits` (lines 27–45) prints the `unsigned` integer values.

# 22.7 Bitwise Operators

```
1  // Fig. 22.11: fig22_11.cpp
2  // Using the bitwise shift operators.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  void displayBits( unsigned ); // prototype
8
9  int main()
10 {
11    unsigned number1 = 960;
12
13    // demonstrate bitwise left shift
14    cout << "The result of left shifting\n";
15    displayBits( number1 );
16    cout << "8 bit positions using the left-shift operator is\n";
17    displayBits( number1 << 8 );
18
19    // demonstrate bitwise right shift
20    cout << "\nThe result of right shifting\n";
21    displayBits( number1 );
22    cout << "8 bit positions using the right-shift operator is\n";
23    displayBits( number1 >> 8 );
24 } // end main
```

**Fig. 22.11** | Bitwise shift operators. (Part 1 of 3.)



# 22.7 Bitwise Operators

```
25
26 // display bits of an unsigned integer value
27 void displayBits( unsigned value )
28 {
29    const int SHIFT = 8 * sizeof( unsigned ) - 1;
30    const unsigned MASK = 1 << SHIFT;
31
32    cout << setw( 10 ) << value << " = ";
33
34    // display bits
35    for ( unsigned i = 1; i <= SHIFT + 1; i++ )
36    {
37       cout << ( value & MASK ? '1' : '0' );
38       value <<= 1; // shift value left by 1
39
40       if ( i % 8 == 0 ) // output a space after 8 bits
41          cout << ' ';
42    } // end for
43
44    cout << endl;
45 } // end function displayBits
```

**Fig. 22.11** | Bitwise shift operators. (Part 2 of 3.)

# 22.7 Bitwise Operators

```
The result of left shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the left-shift operator is
    245760 = 00000000 00000011 11000000 00000000

The result of right shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the right-shift operator is
         3 = 00000000 00000000 00000000 00000011
```

**Fig. 22.11** | Bitwise shift operators. (Part 3 of 3.)



# 22.7 Bitwise Operators

- The left-shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- Bits vacated to the right are replaced with 0s; bits shifted off the left are lost.
- In the program of Fig. 22.11, line 11 assigns variable `number1` the value 960 (00000000 00000000 00000011 11000000).
- The result of left-shifting variable `number1` 8 bits in the expression `number1 << 8` (line 17) is 245760 (00000000 00000011 11000000 00000000).

# 22.7 Bitwise Operators

- The right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand.
- Performing a right shift on an unsigned integer causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost.
- In the program of Fig. 22.11, the result of right-shifting number1 in the expression number1 >> 8 (line 23) is 3 (00000000 00000000 00000000 00000011).



# 22.7 Bitwise Operators

**Common Programming Error 22.5**

*The result of shifting a value is undefined if the right operand is negative or if the right operand is greater than or equal to the number of bits in which the left operand is stored.*

**Portability Tip 22.4**

*The result of right-shifting a signed value is machine dependent. Some machines fill with zeros and others use the sign bit.*

# 22.7 Bitwise Operators

- Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator.
- These bitwise assignment operators are shown in Fig. 22.12; they're used in a similar manner to the arithmetic assignment operators introduced in Chapter 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. 22.12** | Bitwise assignment operators.



# 22.9 Character-Handling Library

- Most data is entered into computers as characters—including letters, digits and various special symbols.
- In this section, we discuss C++'s capabilities for examining and manipulating individual characters.
- In the remainder of the chapter, we continue the discussion of character-string manipulation that we began in Chapter 8.

## 22.9 Character-Handling Library

- The character-handling library includes several functions that perform useful tests and manipulations of character data.
- Each function receives a character—represented as an `int`—or `EOF` as an argument.
- Characters are often manipulated as integers.
- Remember that `EOF` normally has the value `-1` and that some hardware architectures do not allow negative values to be stored in `char` variables.
  - Therefore, the character-handling functions manipulate characters as integers.



## 22.9 Character-Handling Library

- Figure 22.17 summarizes the functions of the character-handling library.
- When using functions from the character-handling library, include the `<cctype>` header file.

# 22.9 Character-Handling Library

| Prototype | Description |
|---|---|
| `int isdigit( int c )` | Returns 1 if c is a digit and 0 otherwise. |
| `int isalpha( int c )` | Returns 1 if c is a letter and 0 otherwise. |
| `int isalnum( int c )` | Returns 1 if c is a digit or a letter and 0 otherwise. |
| `int isxdigit( int c )` | Returns 1 if c is a hexadecimal digit character and 0 otherwise. (See Appendix D, Number Systems, for a detailed explanation of binary, octal, decimal and hexadecimal numbers.) |
| `int islower( int c )` | Returns 1 if c is a lowercase letter and 0 otherwise. |
| `int isupper( int c )` | Returns 1 if c is an uppercase letter; 0 otherwise. |
| `int tolower( int c )` | If c is an uppercase letter, `tolower` returns c as a lowercase letter. Otherwise, `tolower` returns the argument unchanged. |
| `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 1 if c is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and 0 otherwise. |

**Fig. 22.17** | Character-handling library functions. (Part 1 of 2.)



# 22.9 Character-Handling Library

| Prototype | Description |
|---|---|
| `int iscntrl( int c )` | Returns 1 if c is a control character, such as newline (`'\n'`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), vertical tab (`'\v'`), alert (`'\a'`), or backspace (`'\b'`)—and 0 otherwise. |
| `int ispunct( int c )` | Returns 1 if c is a printing character other than a space, a digit, or a letter and 0 otherwise. |
| `int isprint( int c )` | Returns 1 if c is a printing character including space (`' '`) and 0 otherwise. |
| `int isgraph( int c )` | Returns 1 if c is a printing character other than space (`' '`) and 0 otherwise. |

**Fig. 22.17** | Character-handling library functions. (Part 2 of 2.)

# 22.9 Character-Handling Library

- Figure 22.18 demonstrates functions isdigit, isalpha, isalnum and isxdigit.
- Function `isdigit` determines whether its argument is a digit (0–9).
- Function `isalpha` determines whether its argument is an uppercase letter (A–Z) or a lowercase letter (a–z).
- Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit.
- Function `isxdigit` determines whether its argument is a hexadecimal digit (A–F, a–f, 0–9).



# 22.9 Character-Handling Library

```
1  // Fig. 22.18: fig22_18.cpp
2  // Character-handling functions isdigit, isalpha, isalnum and isxdigit.
3  #include <iostream>
4  #include <cctype> // character-handling function prototypes
5  using namespace std;
6
7  int main()
8  {
9     cout << "According to isdigit:\n"
10       << ( isdigit( '8' ) ? "8 is a" : "8 is not a" ) << " digit\n"
11       << ( isdigit( '#' ) ? "# is a" : "# is not a" ) << " digit\n";
12
13    cout << "\nAccording to isalpha:\n"
14       << ( isalpha( 'A' ) ? "A is a" : "A is not a" ) << " letter\n"
15       << ( isalpha( 'b' ) ? "b is a" : "b is not a" ) << " letter\n"
16       << ( isalpha( '&' ) ? "& is a" : "& is not a" ) << " letter\n"
17       << ( isalpha( '4' ) ? "4 is a" : "4 is not a" ) << " letter\n";
18
```

**Fig. 22.18** | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 3.)

# 22.9 Character-Handling Library

```
19      cout << "\nAccording to isalnum:\n"
20         << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
21         << " digit or a letter\n"
22         << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
23         << " digit or a letter\n"
24         << ( isalnum( '#' ) ? "# is a" : "# is not a" )
25         << " digit or a letter\n";
26
27      cout << "\nAccording to isxdigit:\n"
28         << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
29         << " hexadecimal digit\n"
30         << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
31         << " hexadecimal digit\n"
32         << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
33         << " hexadecimal digit\n"
34         << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
35         << " hexadecimal digit\n"
36         << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
37         << " hexadecimal digit" << endl;
38   } // end main
```

**Fig. 22.18** | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 2 of 3.)



# 22.9 Character-Handling Library

```
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. 22.18** | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 3 of 3.)

## 22.9 Character-Handling Library

- Figure 22.18 uses the conditional operator (`?:`) with each function to determine whether the string `" is a "` or the string `" is not a "` should be printed in the output for each character tested.
- For example, line 10 indicates that if `'8'` is a digit—i.e., if `isdigit` returns a true (nonzero) value—the string `"8 is a "` is printed.
- If `'8'` is not a digit (i.e., if `isdigit` returns 0), the string `"8 is not a "` is printed.



## 22.10 Pointer-Based String Manipulation Functions

- The string-handling library provides many useful functions for manipulating string data, comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces such as the separate words in a sentence) and determining the length of strings.
- This section presents some common string-manipulation functions of the string-handling library (from the C++ standard library).
- The functions are summarized in Fig. 22.21; then each is used in a live-code example.
- The prototypes for these functions are located in header file `<cstring>`.

# 22.10 Pointer-Based String Manipulation Functions

| Function prototype | Function description |
|---|---|
| `char *strcpy( char *s1, const char *s2 );` | Copies the string `s2` into the character array `s1`. The value of `s1` is returned. |
| `char *strncpy( char *s1, const char *s2, size_t n );` | Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned. |
| `char *strcat( char *s1, const char *s2 );` | Appends the string `s2` to `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned. |
| `char *strncat( char *s1, const char *s2, size_t n );` | Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned. |

**Fig. 22.21** | String-manipulation functions of the string-handling library. (Part 1 of 3.)



# 22.10 Pointer-Based String Manipulation Functions

| Function prototype | Function description |
|---|---|
| `int strcmp( const char *s1, const char *s2 );` | Compares the string `s1` with the string `s2`. The function returns a value of zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively. |
| `int strncmp( const char *s1, const char *s2, size_t n );` | Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if the `n`-character portion of `s1` is equal to, less than or greater than the corresponding `n`-character portion of `s2`, respectively. |

**Fig. 22.21** | String-manipulation functions of the string-handling library. (Part 2 of 3.)

# 22.10 Pointer-Based String Manipulation Functions

| Function prototype | Function description |
|---|---|
| `char *strtok( char *s1, const char *s2 );` | A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text. The string is broken up based on the characters contained in string `s2`. For instance, if we were to break the string `"this:is:a:string"` into tokens based on the character `':'`, the resulting tokens would be `"this"`, `"is"`, `"a"` and `"string"`. Function `strtok` returns only one token at a time—the first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned. |
| `size_t strlen( const char *s );` | Determines the length of string `s`. The number of characters preceding the terminating null character is returned. |

**Fig. 22.21** | String-manipulation functions of the string-handling library. (Part 3 of 3.)



# 22.10 Pointer-Based String Manipulation Functions

- Several functions in Fig. 22.21 contain parameters with data type `size_t`.
- This type is defined in the header file `<cstring>` to be an unsigned integral type such as `unsigned int` or `unsigned long`.

**Common Programming Error 22.8**

*Forgetting to include the `<cstring>` header file when using functions from the string-handling library causes compilation errors.*

# 22.10 Pointer-Based String Manipulation Functions

- Function strcpy copies its second argument—a string—into its first argument—a character array that must be large enough to store the string and its terminating null character, (which is also copied).
- Function strncpy is much like `strcpy`, except that `strncpy` specifies the number of characters to be copied from the string into the array.
- Function `strncpy` does not necessarily copy the terminating null character of its second argument—a terminating null character is written only if the number of characters to be copied is at least one more than the length of the string.



# 22.10 Pointer-Based String Manipulation Functions

**Common Programming Error 22.9**

*When using* `strncpy`*, the terminating null character of the second argument (a* `char *` *string) will not be copied if the number of characters specified by* `strncpy`*'s third argument is not greater than the second argument's length. In that case, a fatal error may occur if you do not manually terminate the resulting* `char *` *string with a null character.*

## 22.10 Pointer-Based String Manipulation Functions

- Figure 22.22 uses `strcpy` (line 13) to copy the entire string in array `x` into array `y` and uses `strncpy` (line 19) to copy the first `14` characters of array `x` into array `z`.
- Line 20 appends a null character (`'\0'`) to array `z`, because the call to `strncpy` in the program does not write a terminating null character.
- (The third argument is less than the string length of the second argument plus one.)



## 22.10 Pointer-Based String Manipulation Functions

```
1  // Fig. 22.22: fig22_22.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4  #include <cstring> // prototypes for strcpy and strncpy
5  using namespace std;
6
7  int main()
8  {
9     char x[] = "Happy Birthday to You"; // string length 21
10    char y[ 25 ];
11    char z[ 15 ];
12
13    strcpy( y, x ); // copy contents of x into y
14
15    cout << "The string in array x is: " << x
16       << "\nThe string in array y is: " << y << '\n';
17
18    // copy first 14 characters of x into z
19    strncpy( z, x, 14 ); // does not copy null character
20    z[ 14 ] = '\0'; // append '\0' to z's contents
21
22    cout << "The string in array z is: " << z << endl;
23 } // end main
```

**Fig. 22.22** | `strcpy` and `strncpy`. (Part 1 of 2.)

# 22.10 Pointer-Based String Manipulation Functions

```
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. 22.22** | strcpy and strncpy. (Part 2 of 2.)



# 22.10 Pointer-Based String Manipulation Functions

- Function strcat appends its second argument (a string) to its first argument (a character array containing a string).
- The first character of the second argument replaces the null character (`'\0'`) that terminates the string in the first argument.
- You must ensure that the array used to store the first string is large enough to store the combination of the first string, the second string and the terminating null character (copied from the second string).
- Function strncat appends a specified number of characters from the second string to the first string and appends a terminating null character to the result.
- The program of Fig. 22.23 demonstrates function `strcat` (lines 15 and 25) and function `strncat` (line 20).

# 22.10 Pointer-Based String Manipulation Functions

```
1  // Fig. 22.23: fig23_23.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4  #include <cstring> // prototypes for strcat and strncat
5  using namespace std;
6
7  int main()
8  {
9     char s1[ 20 ] = "Happy "; // length 6
10    char s2[] = "New Year "; // length 9
11    char s3[ 40 ] = "";
12
13    cout << "s1 = " << s1 << "\ns2 = " << s2;
14
15    strcat( s1, s2 ); // concatenate s2 to s1 (length 15)
16
17    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
18
19    // concatenate first 6 characters of s1 to s3
20    strncat( s3, s1, 6 ); // places '\0' after last character
21
```

**Fig. 22.23** | strcat and strncat. (Part 1 of 2.)



# 22.10 Pointer-Based String Manipulation Functions

```
22    cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
23       << "\ns3 = " << s3;
24
25    strcat( s3, s1 ); // concatenate s1 to s3
26    cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
27       << "\ns3 = " << s3 << endl;
28 } // end main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

**Fig. 22.23** | strcat and strncat. (Part 2 of 2.)

# 22.10 Pointer-Based String Manipulation Functions

- Figure 22.24 compares three strings using `strcmp` (lines 15–17) and `strncmp` (lines 20–22).
- Function `strcmp` compares its first string argument with its second string argument character by character.
- The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string.
- Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares up to a specified number of characters.
- Function `strncmp` stops comparing characters if it reaches the null character in one of its string arguments.
- The program prints the integer value returned by each function call.



# 22.10 Pointer-Based String Manipulation Functions

**Common Programming Error 22.10**

*Assuming that `strcmp` and `strncmp` return one (a true value) when their arguments are equal is a logic error. Both functions return zero (C++'s false value) for equality. Therefore, when testing two strings for equality, the result of the `strcmp` or `strncmp` function should be compared with zero to determine whether the strings are equal.*

# 22.10 Pointer-Based String Manipulation Functions

```
1  // Fig. 22.24: fig22_24.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstring> // prototypes for strcmp and strncmp
6  using namespace std;
7
8  int main()
9  {
10    char *s1 = "Happy New Year";
11    char *s2 = "Happy New Year";
12    char *s3 = "Happy Holidays";
13
14    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
15       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
16       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
17       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
18
19    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
20       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
21       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
22       << strncmp( s3, s1, 7 ) << endl;
23 } // end main
```

**Fig. 22.24** | strcmp and strncmp. (Part 1 of 2.)



# 22.10 Pointer-Based String Manipulation Functions

```
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. 22.24** | strcmp and strncmp. (Part 2 of 2.)

## 22.10 Pointer-Based String Manipulation Functions

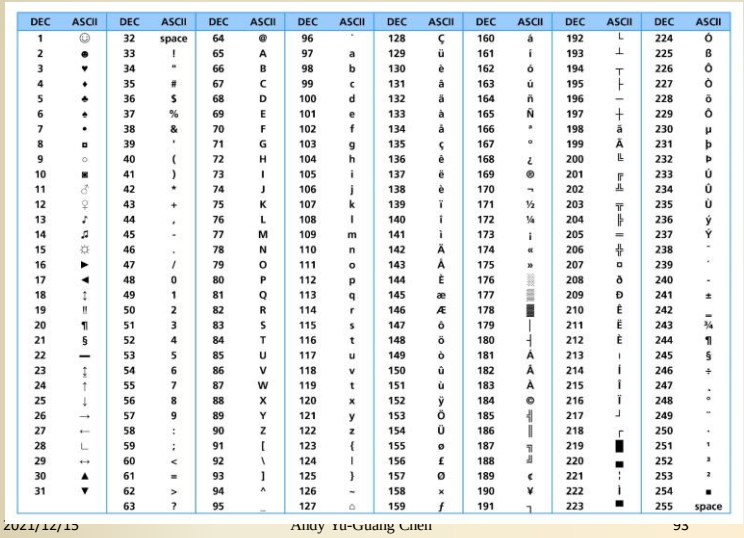
- To understand what it means for one string to be "greater than" or "less than" another, consider the process of alphabetizing last names.
- You'd, no doubt, place "Jones" before "Smith," because the first letter of "Jones" comes before the first letter of "Smith" in the alphabet.
- But the alphabet is more than just a list of 26 letters—it's an ordered list of characters.
- Each letter occurs in a specific position within the list.
- "Z" is more than just a letter of the alphabet; "Z" is specifically the 26th letter of the alphabet.



## 22.10 Pointer-Based String Manipulation Functions

- How does the computer know that one letter comes before another?
- All characters are represented inside the computer as numeric codes; when the computer compares two strings, it actually compares the numeric codes of the characters in the strings.
- In an effort to standardize character representations, most computer manufacturers have designed their machines to utilize one of two popular coding schemes—ASCII or EBCDIC.
- Recall that ASCII stands for "American Standard Code for Information Interchange."
- EBCDIC stands for "Extended Binary Coded Decimal Interchange Code."
- ASCII and EBCDIC are called character codes, or character sets.

# 22.10 Pointer-Based String Manipulation Functions



| DEC | ASCII | DEC | ASCII | DEC | ASCII | DEC | ASCII | DEC | ASCII | DEC | ASCII | DEC | ASCII | DEC | ASCII |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ☺ | 32 | space | 64 | @ | 96 | ` | 128 | Ç | 160 | á | 192 | └ | 224 | Ó |
| 2 | ☻ | 33 | ! | 65 | A | 97 | a | 129 | ü | 161 | í | 193 | ┴ | 225 | ß |
| 3 | ♥ | 34 | " | 66 | B | 98 | b | 130 | é | 162 | ó | 194 | ┬ | 226 | Ô |
| 4 | ♦ | 35 | # | 67 | C | 99 | c | 131 | â | 163 | ú | 195 | ├ | 227 | Ò |
| 5 | ♣ | 36 | $ | 68 | D | 100 | d | 132 | ä | 164 | ñ | 196 | ─ | 228 | õ |
| 6 | ♠ | 37 | % | 69 | E | 101 | e | 133 | à | 165 | Ñ | 197 | ┼ | 229 | Õ |
| 7 | • | 38 | & | 70 | F | 102 | f | 134 | å | 166 | ª | 198 | ã | 230 | µ |
| 8 | ◘ | 39 | ' | 71 | G | 103 | g | 135 | ç | 167 | º | 199 | Ã | 231 | þ |
| 9 | ○ | 40 | ( | 72 | H | 104 | h | 136 | ê | 168 | ¿ | 200 | ╚ | 232 | Þ |
| 10 | ◙ | 41 | ) | 73 | I | 105 | i | 137 | ë | 169 | ® | 201 | ╔ | 233 | Ú |
| 11 | ♂ | 42 | * | 74 | J | 106 | j | 138 | è | 170 | ¬ | 202 | ╩ | 234 | Û |
| 12 | ♀ | 43 | + | 75 | K | 107 | k | 139 | ï | 171 | ½ | 203 | ╦ | 235 | Ù |
| 13 | ♪ | 44 | , | 76 | L | 108 | l | 140 | î | 172 | ¼ | 204 | ╠ | 236 | ý |
| 14 | ♫ | 45 | - | 77 | M | 109 | m | 141 | ì | 173 | ¡ | 205 | ═ | 237 | Ý |
| 15 | ☼ | 46 | . | 78 | N | 110 | n | 142 | Ä | 174 | « | 206 | ╬ | 238 | ¯ |
| 16 | ► | 47 | / | 79 | O | 111 | o | 143 | Å | 175 | » | 207 | ¤ | 239 | ´ |
| 17 | ◄ | 48 | 0 | 80 | P | 112 | p | 144 | É | 176 | ░ | 208 | ð | 240 |  |
| 18 | ↕ | 49 | 1 | 81 | Q | 113 | q | 145 | æ | 177 | ▒ | 209 | Ð | 241 | ± |
| 19 | ‼ | 50 | 2 | 82 | R | 114 | r | 146 | Æ | 178 | ▓ | 210 | Ê | 242 | ‗ |
| 20 | ¶ | 51 | 3 | 83 | S | 115 | s | 147 | ô | 179 | │ | 211 | Ë | 243 | ¾ |
| 21 | § | 52 | 4 | 84 | T | 116 | t | 148 | ö | 180 | ┤ | 212 | È | 244 | ¶ |
| 22 | ▬ | 53 | 5 | 85 | U | 117 | u | 149 | ò | 181 | Á | 213 | ı | 245 | § |
| 23 | ↨ | 54 | 6 | 86 | V | 118 | v | 150 | û | 182 | Â | 214 | Í | 246 | ÷ |
| 24 | ↑ | 55 | 7 | 87 | W | 119 | t | 151 | ù | 183 | À | 215 | Î | 247 | ¸ |
| 25 | ↓ | 56 | 8 | 88 | X | 120 | x | 152 | ÿ | 184 | © | 216 | Ï | 248 | ° |
| 26 | → | 57 | 9 | 89 | Y | 121 | y | 153 | Ö | 185 | ╣ | 217 | ┘ | 249 | ¨ |
| 27 | ← | 58 | : | 90 | Z | 122 | z | 154 | Ü | 186 | ║ | 218 | ┌ | 250 | · |
| 28 | ∟ | 59 | ; | 91 | [ | 123 | { | 155 | ø | 187 | ╗ | 219 | █ | 251 | ¹ |
| 29 | ↔ | 60 | < | 92 | \ | 124 | \| | 156 | £ | 188 | ╝ | 220 | ▄ | 252 | ³ |
| 30 | ▲ | 61 | = | 93 | ] | 125 | } | 157 | Ø | 189 | ¢ | 221 | ¦ | 253 | ² |
| 31 | ▼ | 62 | > | 94 | ^ | 126 | ~ | 158 | × | 190 | ¥ | 222 | Ì | 254 | ■ |
| | | 63 | ? | 95 | _ | 127 | ⌂ | 159 | ƒ | 191 | ┐ | 223 | ▀ | 255 | space |



# 22.10 Pointer-Based String Manipulation Functions

- Function strtok breaks a string into a series of tokens.
- A token is a sequence of characters separated by delimiting characters (usually spaces or punctuation marks).
- For example, in a line of text, each word can be considered a token, and the spaces separating the words can be considered delimiters.
- Multiple calls to `strtok` are required to break a string into tokens (assuming that the string contains more than one token).

## 22.10 Pointer-Based String Manipulation Functions

- The first call to `strtok` contains two arguments, a string to be tokenized and a string containing characters that separate the tokens (i.e., delimiters).
- Line 16 in Fig. 22.25 assigns to `tokenPtr` a pointer to the first token in `sentence`.
- The second argument, `" "`, indicates that tokens in `sentence` are separated by spaces.
- Function `strtok` searches for the first character in `sentence` that is not a delimiting character (space).
- This begins the first token.
- The function then finds the next delimiting character in the string and replaces it with a null (`'\0'`) character.
- This terminates the current token.



## 22.10 Pointer-Based String Manipulation Functions

- Function `strtok` saves (in a `static` variable) a pointer to the next character following the token in `sentence` and returns a pointer to the current token.
- Subsequent calls to `strtok` to continue tokenizing `sentence` contain `NULL` as the first argument (line 22).
- The `NULL` argument indicates that the call to `strtok` should continue tokenizing from the location in `sentence` saved by the last call to `strtok`.
- Function `strtok` maintains this saved information in a manner that is not visible to you.
- If no tokens remain when `strtok` is called, `strtok` returns `NULL`.

# 22.10 Pointer-Based String Manipulation Functions

```
1  // Fig. 22.25: fig22_25.cpp
2  // Using strtok to tokenize a string.
3  #include <iostream>
4  #include <cstring> // prototype for strtok
5  using namespace std;
6
7  int main()
8  {
9     char sentence[] = "This is a sentence with 7 tokens";
10    char *tokenPtr;
11
12    cout << "The string to be tokenized is:\n" << sentence
13       << "\n\nThe tokens are:\n\n";
14
15    // begin tokenization of sentence
16    tokenPtr = strtok( sentence, " " );
17
18    // continue tokenizing sentence until tokenPtr becomes NULL
19    while ( tokenPtr != NULL )
20    {
21       cout << tokenPtr << '\n';
22       tokenPtr = strtok( NULL, " " ); // get next token
23    } // end while
24
```

**Fig. 22.25** | Using `strtok` to tokenize a string. (Part 1 of 2.)



# 22.10 Pointer-Based String Manipulation Functions

```
25    cout << "\nAfter strtok, sentence = " << sentence << endl;
26 } // end main
```

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

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

**Fig. 22.25** | Using `strtok` to tokenize a string. (Part 2 of 2.)

**Common Programming Error 22.11**

*Not realizing that* `strtok` *modifies the string being tokenized, then attempting to use that string as if it were the original unmodified string is a logic error.*

# 22.10 Pointer-Based String Manipulation Functions

- Function strlen takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length.
- The length is also the index of the null character.
- The program of Fig. 22.26 demonstrates function `strlen`.



# 22.10 Pointer-Based String Manipulation Functions

```
1  // Fig. 22.26: fig22_26.cpp
2  // Using strlen.
3  #include <iostream>
4  #include <cstring> // prototype for strlen
5  using namespace std;
6
7  int main()
8  {
9     char *string1 = "abcdefghijklmnopqrstuvwxyz";
10    char *string2 = "four";
11    char *string3 = "Boston";
12
13    cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
14       << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
15       << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
16       << endl;
17 } // end main
```

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

**Fig. 22.26** | `strlen` returns the length of a `char *` string.

## 22.11 Pointer-Based String-Conversion Functions

- In Section 22.10, we discussed several of C++'s most popular pointer-based string-manipulation functions.
- In the next several sections, we cover the remaining functions, including functions for converting strings to numeric values, functions for searching strings and functions for manipulating, comparing and searching blocks of memory.
- This section presents the pointer-based string-conversion functions from the general-utilities library <cstdlib>.



## 22.11 Pointer-Based String-Conversion Functions

- These functions convert pointer-based strings of characters to integer and floating-point values.
- In new code development, C++ programmers typically use the string stream processing capabilities introduced in Chapter 18 to perform such conversions.
- Figure 22.27 summarizes the pointer-based string-conversion functions.
- When using functions from the general-utilities library, include the `<cstdlib>` header file.

# 22.11 Pointer-Based String-Conversion Functions

| Prototype | Description |
|---|---|
| `double atof( const char *nPtr )` | Converts the string `nPtr` to `double`. If the string cannot be converted, `0` is returned. |
| `int atoi( const char *nPtr )` | Converts the string `nPtr` to `int`. If the string cannot be converted, `0` is returned. |
| `long atol( const char *nPtr )` | Converts the string `nPtr` to `long int`. If the string cannot be converted, `0` is returned. |
| `double strtod( const char *nPtr, char **endPtr )` | Converts the string `nPtr` to `double`. `endPtr` is the address of a pointer to the rest of the string after the `double`. If the string cannot be converted, `0` is returned. |

**Fig. 22.27** | Pointer-based string-conversion functions of the general-utilities library. (Part 1 of 2.)



# 22.11 Pointer-Based String-Conversion Functions

| Prototype | Description |
|---|---|
| `long strtol( const char *nPtr, char **endPtr, int base )` | Converts the string `nPtr` to `long`. `endPtr` is the address of a pointer to the rest of the string after the `long`. If the string cannot be converted, `0` is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal. |
| `unsigned long strtoul( const char *nPtr, char **endPtr, int base )` | Converts the string `nPtr` to `unsigned long`. `endPtr` is the address of a pointer to the rest of the string after the `unsigned long`. If the string cannot be converted, `0` is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal. |

**Fig. 22.27** | Pointer-based string-conversion functions of the general-utilities library. (Part 2 of 2.)

# 22.11 Pointer-Based String-Conversion Functions

- Function atoi (Fig. 22.29, line 9) converts its argument—a string of digits that represents an integer—to an `int` value.
- The function returns the `int` value.
- If the string cannot be converted, function `atoi` returns zero.



# 22.11 Pointer-Based String-Conversion Functions

```
1  // Fig. 22.29: Fig22_29.cpp
2  // Using atoi.
3  #include <iostream>
4  #include <cstdlib> // atoi prototype
5  using namespace std;
6
7  int main()
8  {
9     int i = atoi( "2593" ); // convert string to int
10
11    cout << "The string \"2593\" converted to int is " << i
12       << "\nThe converted value minus 593 is " << i - 593 << endl;
13 } // end main
```

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

**Fig. 22.29** | String-conversion function `atoi`.

# 22.12 Search Functions of the Pointer-Based String-Handling Library

- This section presents the functions of the string-handling library used to search strings for characters and other strings.
- The functions are summarized in Fig. 22.34.
- Functions `strcspn` and `strspn` specify return type `size_t`.
- Type `size_t` is a type defined by the standard as the integral type of the value returned by operator `sizeof`



# 22.12 Search Functions of the Pointer-Based String-Handling Library

| 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. |
| `char *strrchr( const char *s, int c )` | Searches from the end of string `s` and locates the last occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in string `s` is returned. Otherwise, a null pointer is returned. |
| `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 )` | 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. |

**Fig. 22.34** | Search functions of the pointer-based string-handling library. (Part 1 of 2.)

# 22.12 Search Functions of the Pointer-Based String-Handling Library

| Prototype | Description |
|---|---|
| `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`. |
| `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. 22.34** | Search functions of the pointer-based string-handling library. (Part 2 of 2.)



# 22.13 Memory Functions of the Pointer-Based String-Handling Library

- The string-handling library functions presented in this section facilitate manipulating, comparing and searching blocks of memory.
- The functions treat blocks of memory as arrays of bytes.
- These functions can manipulate any block of data.
- Figure 22.41 summarizes the memory functions of the string-handling library.
- In the function discussions, “object” refers to a block of data.

# 22.13 Memory Functions of the Pointer-Based String-Handling Library

| 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. The area from which characters are copied is not allowed to overlap the area to which characters are copied. |
| `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 were first copied from the object pointed to by `s2` into a temporary array, then copied from the temporary array into the object pointed to by `s1`. A pointer to the resulting object is returned. The area from which characters are copied is allowed to overlap the area to which characters are copied. |
| `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`, respectively. |

**Fig. 22.41** | Memory functions of the string-handling library. (Part 1 of 2.)



# 22.13 Memory Functions of the Pointer-Based String-Handling Library

| Prototype | Description |
|---|---|
| `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. 22.41** | Memory functions of the string-handling library. (Part 2 of 2.)

# Summary

- Create and use struct
- Pass struct to function
- Use typedef
- Bitwise operator
- Character-Handling Library
- Pointer-Based String Manipulation Functions
- Pointer-Based String-Conversion Functions
- Search Functions of the Pointer-Based String-Handling Library
- Memory Functions of the Pointer-Based String-Handling Library





**Andy, Yu-Guang Chen**

**Assistant Professor, Department of EE, NCU**

**Email: andyygchen@ee.ncu.edu.tw**