Lecture 20:
Bits, Characters, and Structs

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 30th, 2010
• Structures are aggregate data types—that is, they can be built using elements of several types including other structs.

• Consider the following structure definition:

  ```
  struct Card {
    string face;
    string suit;
  }; // end struct Card
  ```

  – Keyword `struct` introduces the definition for structure `Card`.
  – The identifier `Card` is the structure name and is used in C++ to declare variables of the structure type (in C, the type name of the preceding structure is `struct Card`).
  – In this example, the structure type is `Card`.
  – Data (and possibly functions—just as with classes) declared within the braces of the structure definition are the structure’s members.
• 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.
21.2 Structure Definitions (cont.)

- Structure members can be variables of the fundamental data types (e.g., `int`, `double`, etc.) or aggregates, such as arrays, other structures and classes.
- Data members in a single structure definition can be of many data types.
- A structure cannot contain an instance of itself.
  - A pointer to a structure of the same type, however, can be included.
  - A structure containing a member that is a pointer to the same structure type is referred to as a self-referential structure.
  - We can use self-referential classes to build various kinds of linked data structures.
21.2 Structure Definitions (cont.)

- 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 only valid built-in operations that may be performed on structure objects are
  – assigning one structure object to another of the same type,
  – taking the address (&) of a structure object,
  – accessing the members of a structure object (in the same manner as members of a class are accessed) and
  – using the `sizeof` operator to determine the size of a structure.
• 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.
Consider the following structure definition in which structure objects `sample1` and `sample2` of type `Example` are declared:

```c
• struct Example
  
  {  
      char c;
      int i;
  }

• 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).
Fig. 21.1 | Possible storage alignment for a variable of type Example, showing an undefined area in memory.
Structures can be initialized using initializer lists, like arrays.
For example, the declaration
   • Card oneCard = { "Three", "Hearts" };
creates Card variable oneCard and initializes member face
to "Three" and member suit to "Hearts".
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.
21.4 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.
- 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.
21.7 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.
• 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.
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.
21.7 Bitwise Operators (cont.)

• 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.
• 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.
When using the bitwise operators, it’s useful to illustrate their precise effects by printing values in their binary representation.

The program of Fig. 21.6 prints an unsigned integer in its binary representation in groups of eight bits each.
21.7 Bitwise Operators (cont.)

// Fig. 21.6: fig21_06.cpp
// Printing an unsigned integer in bits.
#include <iostream>
#include <iomanip>
using namespace std;

void displayBits( unsigned ); // prototype

int main()
{
    unsigned inputValue; // integral value to print in binary

cout << "Enter an unsigned integer: ";
cin >> inputValue;
displayBits( inputValue );
} // end main

// display bits of an unsigned integer value
void displayBits( unsigned value )
{
    const int SHIFT = 8 * sizeof( unsigned ) - 1;
    const unsigned MASK = 1 << SHIFT;

Fig. 21.6 | Printing an unsigned integer in bits. (Part 1 of 2.)
cout << setw(10) << value << " = ";

// display bits
for (unsigned i = 1; i <= SHIFT + 1; i++)
{
    cout << (value & MASK ? '1' : '0');
    value <<= 1; // shift value left by 1

    if (i % 8 == 0) // output a space after 8 bits
        cout << ' ';    
    // end for
}

} // end function displayBits

Enter an unsigned integer: 65000
65000 = 00000000 00000000 11111101 11010000

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

Fig. 21.6  |  Printing an unsigned integer in bits. (Part 2 of 2.)
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`. 
The value of constant `SHIFT` was calculated in line 21 with the expression

\[ 8 \times \text{sizeof}(\text{unsigned}) - 1 \]

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 \( 1 \ll \text{SHIFT} \) 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.
• 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
  
  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \hline
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
  \end{array}
  \]

  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`. 

©1992-2010 by Pearson Education, Inc. All Rights Reserved.
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)

\[
\begin{array}{cccc}
\text{10000000} & \text{00000000} & \text{00000000} & \text{00000000} \\
\end{array}
\]

(value & MASK)

Because both left bits are 1s, the expression’s result is nonzero (true) and 1 is printed.

Figure 21.7 summarizes the results of combining two bits with the bitwise AND operator.
Common Programming Error 21.3
Using the logical AND operator (&&) for the bitwise AND operator (&) and vice versa is a logic error.

Common Programming Error 21.4
Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is a logic error.
• 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.”

• For example if variable number1 has the value 21845 (00000000 00000000 01010101 01010101).

• When the expression ~number1 evaluates, the result is (11111111 11111111 10101010 10101010).

• Figure 21.11 demonstrates the left-shift operator (<<) and the right-shift operator (>>).

• Function displayBits (lines 27–45) prints the unsigned integer values.
21.7 Bitwise Operators (cont.)

```cpp
// Fig. 21.11: fig21_11.cpp
// Using the bitwise shift operators.
#include <iostream>
#include <iomanip>
using namespace std;

void displayBits( unsigned ); // prototype

int main()
{
    unsigned number1 = 960;

    // demonstrate bitwise left shift
    cout << "The result of left shifting\n";
    displayBits( number1 );
    cout << "8 bit positions using the left-shift operator is\n";
    displayBits( number1 << 8 );

    // demonstrate bitwise right shift
    cout << "\nThe result of right shifting\n";
    displayBits( number1 );
    cout << "8 bit positions using the right-shift operator is\n";
    displayBits( number1 >> 8 );
} // end main
```

Fig. 21.11 | Bitwise shift operators. (Part 1 of 3.)
21.7 Bitwise Operators (cont.)

```cpp
// display bits of an unsigned integer value
void displayBits( unsigned value )
{
    const int SHIFT = 8 * sizeof( unsigned ) - 1;
    const unsigned MASK = 1 << SHIFT;

    cout << setw( 10 ) << value << " = ";

    // display bits
    for ( unsigned i = 1; i <= SHIFT + 1; i++ )
    {
        cout << ( value & MASK ? '1' : '0' );
        value <<= 1; // shift value left by 1

        if ( i % 8 == 0 ) // output a space after 8 bits
            cout << ' ';
    } // end for

    cout << endl;
} // end function displayBits
```

**Fig. 21.11** Bitwise shift operators. (Part 2 of 3.)
21.7 Bitwise Operators (cont.)

The result of left shifting

960 = 00000000 00000000 00000111 11000000

8 bit positions using the left-shift operator is

245760 = 00000000 00000111 11000000 00000000

The result of right shifting

960 = 00000000 00000000 00000111 11000000

8 bit positions using the right-shift operator is

3 = 00000000 00000000 00000000 00000011

**Fig. 21.11** Bitwise shift operators. (Part 3 of 3.)
• 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. 21.11, line 11 assigns variable `number1` the value 960 (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).
• 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. 21.11, the result of right-shifting `number1` in the expression `number1 >> 8` (line 23) is 3 (00000000 00000000 00000000 00000011).
21.8 Bit Fields

• C++ provides the ability to specify the number of bits in which an integral type or `enum` type member of a class or a structure is stored.

• Such a member is referred to as a `bit field`.

• Bit fields enable better memory utilization by storing data in the minimum number of bits required.

• Bit field members must be declared as an integral or `enum` type.
21.8 Bit Fields

**Performance Tip 21.2**

*Bit fields help conserve storage.*
• `struct BitCard`  
  {
    `unsigned face : 4;`
    `unsigned suit : 2;`
    `unsigned color : 1;`
  }  // end struct BitCard

• The definition contains three `unsigned` bit fields—`face`, `suit` and `color`—used to represent a card from a deck of 52 cards.

• A bit field is declared by following an integral type or `enum` type member with a colon (`:`) and an integer constant representing the width of the bit field (i.e., the number of bits in which the member is stored).

• The width must be an integer constant.

• The preceding structure definition indicates that member `face` is stored in 4 bits, member `suit` in 2 bits and member `color` in 1 bit.
The number of bits is based on the desired range of values for each structure member.

- Member `face` stores values between 0 (Ace) and 12 (King)—4 bits can store a value between 0 and 15.
- Member `suit` stores values between 0 and 3 (0 = Diamonds, 1 = Hearts, 2 = Clubs, 3 = Spades)—2 bits can store a value between 0 and 3.
- Finally, member `color` stores either 0 (Red) or 1 (Black)—1 bit can store either 0 or 1.
21.8 Bit Fields (cont.)

// Fig. 21.14: DeckOfCards.h
// Definition of class DeckOfCards that
// represents a deck of playing cards.
#include <vector>
using namespace std;

// BitCard structure definition with bit fields
struct BitCard
{
    unsigned face : 4; // 4 bits; 0-15
    unsigned suit : 2; // 2 bits; 0-3
    unsigned color : 1; // 1 bit; 0-1
}; // end struct BitCard

// DeckOfCards class definition
class DeckOfCards
{
    public:
    static const int faces = 13;
    static const int colors = 2; // black and red
    static const int numberOfCards = 52;

Fig. 21.14 | Header file for class DeckOfCards. (Part 1 of 2.)
21.8 Bit Fields (cont.)

// Fig. 21.15: DeckOfCards.cpp
// Member-function definitions for class DeckOfCards that simulates
// the shuffling and dealing of a deck of playing cards.
#include <iostream>
#include <iomanip>
#include "DeckOfCards.h" // DeckOfCards class definition
using namespace std;

// no-argument DeckOfCards constructor initializes deck
DeckOfCards::DeckOfCards()
{
    for ( int i = 0; i < numberOfCards; i++ )
    {
        deck[ i ].face = i % faces; // faces in order
        deck[ i ].suit = i / faces; // suits in order
        deck[ i ].color = i / ( faces * colors ); // colors in order
    } // end for
} // end no-argument DeckOfCards constructor

Fig. 21.15 | Class file for DeckOfCards. (Part 1 of 2.)
21.8 Bit Fields (cont.)

```cpp
21    // deal cards in deck
22    void DeckOfCards::deal()
23    {
24        for ( int k1 = 0, k2 = k1 + numberOfCards / 2;
25            k1 < numberOfCards / 2 - 1; k1++, k2++)
26            cout << "Card:" << setw(3) << deck[ k1 ].face
27                << " Suit:" << setw(2) << deck[ k1 ].suit
28                << " Color:" << setw(2) << deck[ k1 ].color
29                << " " << "Card:" << setw(3) << deck[ k2 ].face
30                << " Suit:" << setw(2) << deck[ k2 ].suit
31                << " Color:" << setw(2) << deck[ k2 ].color << endl;
32    } // end function deal
```

**Fig. 21.15**  Class file for DeckOfCards. (Part 2 of 2.)
### 21.8 Bit Fields (cont.)

<table>
<thead>
<tr>
<th>Card</th>
<th>Suit</th>
<th>Color</th>
<th>Card</th>
<th>Suit</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**Fig. 21.16** Bit fields used to store a deck of cards. (Part 2 of 3.)
Questions