

A decorative vertical bar on the left side of the slide. It consists of a dark teal background with a white dotted pattern. Overlaid on this are several orange circles of varying sizes, arranged in a cluster. The largest circle is at the top left, with smaller ones below and to its right. The text "OBJECT ORIENTED PROGRAMMING USING C++" is centered in the upper half of the slide.

# OBJECT ORIENTED PROGRAMMING USING C++

# Chapter 10 - Structures, Unions, Bit Manipulations, and Enumerations

## Outline

- 10.1 Introduction
- 10.2 Structure Definitions
- 10.3 Initializing Structures
- 10.4 Accessing Members of Structures
- 10.5 Using Structures with Functions
- 10.6 Typedef
- 10.7 Example: High-Performance Card Shuffling and Dealing Simulation
- 10.8 Unions
- 10.9 Bitwise Operators
- 10.10 Bit Fields
- 10.11 Enumeration Constants



## 10.1 Introduction

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - Combined with pointers, can create linked lists, stacks, queues, and trees



## 10.2 Structure Definitions

- Example

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

- **struct** introduces the definition for structure **card**
- **card** is the *structure name* and is used to declare variables of the *structure type*
- **card** contains two members of type **char \*** - **face** and **suit**



## 10.2 Structure Definitions (II)

- Struct information
  - A struct cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - Structure definition does not reserve space in memory
  - Creates a new data type that used to declare structure variables.
- Declarations
  - Declared like other variables:

```
card oneCard, deck[ 52 ], *cPtr;
```
  - Can use a comma separated list:

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



## 10.2 Structure Definitions (III)

- 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



## 10.3 Initializing Structures

- Initializer lists

- Example:

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

- Assignment statements

- Example:

```
card threeHearts = oneCard;
```

- Or:

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



## 10.4 Accessing Members of Structures

- Accessing structure members

- Dot operator (.) - use with structure variable name

```
card myCard;  
printf( "%s", myCard.suit );
```

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

```
card *myCardPtr = &myCard;  
printf( "%s", myCardPtr->suit );
```

`myCardPtr->suit` equivalent to `( *myCardPtr ).suit`





## 10.5 Using Structures With Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure



## 10.6 Typedef

- **typedef**
  - Creates synonyms (aliases) for previously defined data types
  - Use **typedef** to create shorter type names.
  - Example:

```
typedef Card *CardPtr;
```
  - Defines a new type name **CardPtr** as a synonym for type **Card \***
  - **typedef** does not create a new data type
    - Only creates an alias



## 10.7 Example: High-Performance Card-shuffling and Dealing Simulation

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



```

1  /* Fig. 10.3: fig10_03.c
2     The card shuffling and dealing program using structures */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  struct card {
8     const char *face;
9     const char *suit;
10 };
11
12 typedef struct card Card;
13
14 void fillDeck( Card * const, const char *[],
15              const char *[] );
16 void shuffle( Card * const );
17 void deal( const Card * const );
18
19 int main()
20 {
21     Card deck[ 52 ];
22     const char *face[] = { "Ace", "Deuce", "Three",
23                          "Four", "Five",
24                          "Six", "Seven", "Eight",
25                          "Nine", "Ten",
26                          "Jack", "Queen", "King"};
27     const char *suit[] = { "Hearts", "Diamonds",
28                          "Clubs", "Spades"};
29
30     srand( time( NULL ) );

```



## Outline



### 1. Load headers

#### 1.1 Define struct

#### 1.2 Function prototypes

#### 1.3 Initialize deck[ ] and face[ ]

#### 1.4 Initialize suit[ ]



## Outline

```
31
32  fillDeck( deck, face, suit );
33  shuffle( deck );
34  deal( deck );
35  return 0;
36 }
37
38 void fillDeck( Card * const wDeck, const char * wFace[],
39              const char * wSuit[] )
40 {
41     int i;
42
43     for ( i = 0; i <= 51; i++ ) {
44         wDeck[ i ].face = wFace[ i % 13 ];
45         wDeck[ i ].suit = wSuit[ i / 13 ];
46     }
47 }
48
49 void shuffle( Card * const wDeck )
50 {
51     int i, j;
52     Card temp;
53
54     for ( i = 0; i <= 51; i++ ) {
55         j = rand() % 52;
56         temp = wDeck[ i ];
57         wDeck[ i ] = wDeck[ j ];
58         wDeck[ j ] = temp;
59     }
60 }
```

Put all 52 cards in the deck.  
**face** and **suit** determined by  
remainder (modulus).

## 2. Randomize

### 2. fillDeck

### 2.2 deal

## 3. Function definitions

Select random number between 0 and 51.  
Swap element **i** with that element.

```
61
62 void deal( const Card * const wDeck )
63 {
64     int i;
65
66     for ( i = 0; i <= 51; i++ )
67         printf( "%5s of %-8s%c", wDeck[ i ].face,
68                 wDeck[ i ].suit,
69                 ( i + 1 ) % 2 ? '\t' : '\n' );
70 }
```



## Outline

Cycle through array and print out data.

S

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



## Outline

### Program Output

## 10.8 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
- Only the last data member defined can be accessed

- **union declarations**

- Same as **struct**

```
union Number {  
    int x;  
    float y;  
};  
Union myObject;
```





## 10.8 Unions (II)

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



```

1  /* Fig. 10.5: fig10_05.c
2     An example of a union */
3  #include <stdio.h>
4
5  union number {
6     int x;
7     double y;
8  };
9
10 int main()
11 {
12     union number value;
13
14     value.x = 100;
15     printf( "%s\n%s\n%s%d\n%s%f\n\n",
16             "Put a value in the integer member",
17             "and print both members.",
18             "int:  ", value.x,
19             "double:\n", value.y );
20
21     value.y = 100.0;
22     printf( "%s\n%s\n%s%d\n%s%f\n",
23             "Put a value in the floating member",
24             "and print both members.",
25             "int:  ", value.x,
26             "double:\n", value.y );
27     return 0;
28 }

```



## Outline

### 1. Define union

#### 1.1 Initialize variables

### 2. Set variables

### 3. Print

Put a value in the integer member  
and print both members.

int: 100

double:

-9255959211743313600.00000

Put a value in the floating member  
and print both members.

int: 0

double:

100.000000



Outline

**Program Output**

## 10.9 Bitwise Operators

- All data represented internally as sequences of bits
  - Each bit can be either **0** or **1**
  - Sequence of 8 bits forms a *byte*

Operator	Name	Description
&	bitwise AND	The bits in the result are set to <b>1</b> if the corresponding bits in the two operands are both <b>1</b> .
	bitwise OR	The bits in the result are set to <b>1</b> if at least one of the corresponding bits in the two operands is <b>1</b> .
^	bitwise exclusive OR	The bits in the result are set to <b>1</b> if exactly one of the corresponding bits in the two operands is <b>1</b> .
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with <b>0</b> bits.
>>	right shift	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 <b>0</b> bits are set to <b>1</b> and all <b>1</b> bits are set to <b>0</b> .



```

1  /* Fig. 10.9: fig10_09.c
2     Using the bitwise AND, bitwise inclusive OR, bitwise
3     exclusive OR and bitwise complement operators */
4  #include <stdio.h>
5
6  void displayBits( unsigned );
7
8  int main()
9  {
10     unsigned number1, number2, mask, setBits;
11
12     number1 = 65535;
13     mask = 1;
14     printf( "The result of combining the following\n" );
15     displayBits( number1 );
16     displayBits( mask );
17     printf( "using the bitwise AND operator & is\n" );
18     displayBits( number1 & mask );
19
20     number1 = 15;
21     setBits = 241;
22     printf( "\nThe result of combining the following\n" );
23     displayBits( number1 );
24     displayBits( setBits );
25     printf( "using the bitwise inclusive OR operator | is\n" );
26     displayBits( number1 | setBits );
27
28     number1 = 139;
29     number2 = 199;
30     printf( "\nThe result of combining the following\n" );

```



## Outline

### 1. Function prototype

#### 1.1 Initialize variables

### 2. Function calls

#### 2.1 Print

```

31  displayBits( number1 );
32  displayBits( number2 );
33  printf( "using the bitwise exclusive OR operator ^ is\n" );
34  displayBits( number1 ^ number2 );
35
36  number1 = 21845;
37  printf( "\nThe one's complement of\n" );
38  displayBits( number1 );
39  printf( "is\n" );
40  displayBits( ~number1 );
41
42  return 0;
43 }
44
45 void displayBits( unsigned value )
46 {
47     unsigned c, displayMask = 1 << 31;
48
49     printf( "%7u = ", value );
50
51     for ( c = 1; c <= 32; c++ ) {
52         putchar( value & displayMask ? '1' : '0' );
53         value <<= 1;
54
55         if ( c % 8 == 0 )
56             putchar( ' ' );
57     }
58
59     putchar( '\n' );
60 }

```



## Outline

### 2.1 Print

### 3. Function definition

**MASK** created with only one set bit  
i.e. (10000000 00000000)

The **MASK** is constantly **AND**ed with **value**.  
**MASK** only contains one bit, so if the **AND** returns true it means **value** must have that bit.  
**value** is then shifted to test the next bit.

```
The result of combining the following
 65535 = 00000000 00000000 11111111 11111111
  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
```



Outline

**Program Output**

## 10.10 Bit Fields

- Bit field
  - Member of a structure whose size (in bits) has been specified
  - Enable better memory utilization
  - *Must* be declared as **int** or **unsigned**
  - Cannot access individual bits
- Declaring bit fields
  - Follow **unsigned** or **int** member with a colon (:) and an integer constant representing the *width* of the field
  - Example:

```
struct BitCard {  
    unsigned face : 4;  
    unsigned suit : 2;  
    unsigned color : 1;  
};
```





## 10.10 Bit Fields (II)

- Unnamed bit field

- Field used as padding in the structure
- Nothing may be stored in the bits

```
struct Example {  
    unsigned a : 13;  
    unsigned   : 3;  
    unsigned b : 4;  
}
```

- Unnamed bit field with zero width aligns next bit field to a new storage unit boundary



## 10.11 Example: A Game of Chance and Introducing enum

- Enumeration
  - Set of integers represented by identifiers
  - Enumeration constants - like symbolic constants whose values automatically set
    - Values start at **0** and are incremented by **1**
    - Values can be set explicitly with **=**
    - Need unique constant names
  - Declare variables as normal
    - Enumeration variables can *only* assume their enumeration constant values (not the integer representations)



## 10.11 Example: A Game of Chance and Introducing enum (II)

- Example:

```
enum Months { JAN = 1, FEB, MAR, APR, MAY,  
             JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

- Starts at 1, increments by 1



```
1 /* Fig. 10.18: fig10_18.c
2    Using an enumeration type */
3 #include <stdio.h>
4
5 enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
6              JUL, AUG, SEP, OCT, NOV, DEC };
7
8 int main()
9 {
10    enum months month;
11    const char *monthName[] = { "", "January", "February",
12                               "March", "April", "May",
13                               "June", "July", "August",
14                               "September", "October",
15                               "November", "December" };
16
17    for ( month = JAN; month <= DEC; month++ )
18        printf( "%2d%11s\n", month, monthName[ month ] );
19
20    return 0;
21 }
```



## Outline



### 1. Define enumeration

#### 1.1 Initialize variable

### 2. Loop

#### 2.1 Print

- 1 January
- 2 February
- 3 March
- 4 April
- 5 May
- 6 June
- 7 July
- 8 August
- 9 September
- 10 October
- 11 November
- 12 December



## Outline

### **Program Output**