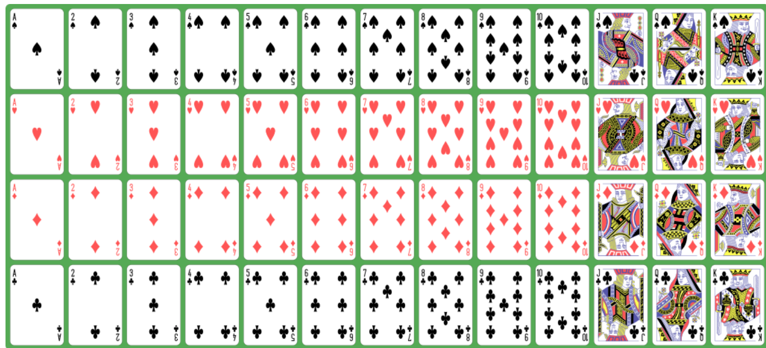
CC-112 Programming Fundamentals

Structures

Nazar Khan
Department of Computer Science
University of the Punjab

How can you represent a card?



- 4 *suits* – spade, heart, diamond, club
- 13 *faces* per suit – A, 2, . . . , 10, J, Q, K
- Each card has 2 attributes – a suit and a face.
- Just like `int`, `float`, `double`, `...`, can we have a *data type for storing cards?*

Structure

- *Structures* are derived data types – they are constructed using objects of other types.
- An array contains items of the same type.
- A structure can contain items of different types.

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

struct card aCard, deck[52], *cardPtr;
struct card bCard = { "Three", "Hearts" };
struct card cCard = bCard;
```

Structure

- A structure can not contain as a member an object of the same structure.
- But it can contain a pointer to another object of the same structure.

```
struct employee {
  char firstName[20];
  char lastName[20];
  unsigned int age;
  char gender;
  double hourlySalary;
  struct employee teamLeader; //ERROR
  struct employee *teamLeaderPtr; //pointer
};
```

Accessing members of a structure

- `aCard.face` when `aCard` is a *struct object*.
- `cardPtr->face` when `cardPtr` is a *struct pointer*.

```
// Structure member operator and structure pointer operator
#include <stdio.h>
// card structure definition
struct card {
   char *face; // define pointer face
   char *suit; // define pointer suit
};
int main(void)
{
   struct card aCard; // define one struct card variable
   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";
   struct card *cardPtr = &aCard; // assign address of aCard to cardPtr
   printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
      cardPtr->face, " of ", cardPtr->suit,
      (*cardPtr).face, " of ", (*cardPtr).suit);
}
```

Output

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

typedef

- The keyword typedef provides a mechanism for creating synonyms (or aliases) for previously defined data types.

```
typedef struct card Card;
Card deck[52];
```

is the same as

```
struct card deck[52];
```

High performance card shuffling

Via array of structures and passing by reference

```c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

typedef struct card Card; // new type name for struct card

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

int main(void)
{
   Card deck[CARDS]; // define array of Cards

   // initialize array of pointers
```

High performance card shuffling

Via array of structures and passing by reference

```
   const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King"};

   // initialize array of pointers
   const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

   srand(time(NULL)); // randomize

   fillDeck(deck, face, suit); // load the deck with Cards
   shuffle(deck); // put Cards in random order
   deal(deck); // deal all 52 Cards
}

// place strings into Card structures
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[])
{
   // loop through wDeck
   for (size_t i = 0; i < CARDS; ++i) {
      wDeck[i].face = wFace[i % FACES];
      wDeck[i].suit = wSuit[i / FACES];
   }
}

// shuffle cards
void shuffle(Card * const wDeck)
```

High performance card shuffling

Via array of structures and passing by reference

```
{
   // loop through wDeck randomly swapping Cards
   for (size_t i = 0; i < CARDS; ++i) {
      size_t j = rand() % CARDS;
      Card temp = wDeck[i];
      wDeck[i] = wDeck[j];
      wDeck[j] = temp;
   }
}

// deal cards
void deal(const Card * const wDeck)
{
   // loop through wDeck
   for (size_t i = 0; i < CARDS; ++i) {
      printf("%5s of %-8s%s", wDeck[i].face, wDeck[i].suit,
         (i + 1) % 4 ? "  " : "\n");
   }
}
```

High performance card shuffling

Via array of structures and passing by reference

Output

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

Bitwise Operators

- Computers represent all data internally as sequences of bits.

Operator		Description
&	bitwise AND	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are *both* 1.
|	bitwise inclusive OR	Compares its two operands bit by bit. The bits in the result are set to 1 if *at least one* of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR (also known as bitwise XOR)	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are different.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 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 when the left operand is negative.
~	complement	All 0 bits are set to 1 and all 1 bits are set to 0.

Bitwise AND, OR, XOR and Complement

```
// Using the bitwise AND, bitwise inclusive OR, bitwise
// exclusive OR and bitwise complement operators
#include <stdio.h>
void displayBits(unsigned int value); // prototype
int main(void)
{
   // demonstrate bitwise AND (&)
   unsigned int number1 = 65535;
   unsigned int mask = 1;
   puts("The result of combining the following");
   displayBits(number1);
   displayBits(mask);
   puts("using the bitwise AND operator & is");
   displayBits(number1 & mask);

   // demonstrate bitwise inclusive OR (|)
   number1 = 15;
   unsigned int setBits = 241;
   puts("\nThe result of combining the following");
   displayBits(number1);
   displayBits(setBits);
   puts("using the bitwise inclusive OR operator | is");
   displayBits(number1 | setBits);

   // demonstrate bitwise exclusive OR (^)
   number1 = 139;
   unsigned int number2 = 199;
   puts("\nThe result of combining the following");
```

Bitwise AND, OR, XOR and Complement

```
   displayBits(number1);
   displayBits(number2);
   puts("using the bitwise exclusive OR operator ^ is");
   displayBits(number1 ^ number2);

   // demonstrate bitwise complement (~)
   number1 = 21845;
   puts("\nThe one's complement of");
   displayBits(number1);
   puts("is");
   displayBits(~number1);
}

// display bits of an unsigned int value
void displayBits(unsigned int value)
{
   // declare displayMask and left shift 31 bits
   unsigned int displayMask = 1 << 31;
   printf("%10u = ", value);
   // loop through bits
   for (unsigned int c = 1; c <= 32; ++c) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1; // shift value left by 1
      if (c % 8 == 0) { // output a space after 8 bits
         putchar(' ');
      }
   }
   putchar('\n');
}
```

Bitwise AND, OR, XOR and Complement

Output

```
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
```

Bitwise left- and right-shift

```c
// Using the bitwise shift operators
#include <stdio.h>

void displayBits(unsigned int value); // prototype

int main(void)
{
   unsigned int number1 = 960; // initialize number1

   // demonstrate bitwise left shift
   puts("\nThe result of left shifting");
   displayBits(number1);
   puts("8 bit positions using the left shift operator << is");
   displayBits(number1 << 8);

   // demonstrate bitwise right shift
   puts("\nThe result of right shifting");
   displayBits(number1);
   puts("8 bit positions using the right shift operator >> is");
   displayBits(number1 >> 8);
}

// display bits of an unsigned int value
void displayBits(unsigned int value)
{
   // declare displayMask and left shift 31 bits
   unsigned int displayMask = 1 << 31;
```

Bitwise left- and right-shift

```
   printf("%7u = ", value);

   // loop through bits
   for (unsigned int c = 1; c <= 32; ++c) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1; // shift value left by 1

      if (c % 8 == 0) { // output a space after 8 bits
         putchar(' ');
      }
   }

   putchar('\n');
}
```

Output

```
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
```

Operator Precedences

<table>
<tr><th>Operator</th><th>Associativity</th><th>Type</th></tr>
<tr><td>() [] . -> ++ (postfix) -- (postfix)</td><td>left to right</td><td>highest</td></tr>
<tr><td>+ - ++ -- ! & * ~ sizeof (type)</td><td>right to left</td><td>unary</td></tr>
<tr><td>* / %</td><td>left to right</td><td>multiplicative</td></tr>
<tr><td>+ -</td><td>left to right</td><td>additive</td></tr>
<tr><td><< >></td><td>left to right</td><td>shifting</td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td>relational</td></tr>
<tr><td>== !=</td><td>left to right</td><td>equality</td></tr>
<tr><td>&</td><td>left to right</td><td>bitwise AND</td></tr>
<tr><td>^</td><td>left to right</td><td>bitwise XOR</td></tr>
<tr><td>|</td><td>left to right</td><td>bitwise OR</td></tr>
<tr><td>&&</td><td>left to right</td><td>logical AND</td></tr>
<tr><td>||</td><td>left to right</td><td>logical OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= &= |= ^= <<= >>= %=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>,</td><td>left to right</td><td>comma</td></tr>
</table>

Bit fields for efficient memory utilization

```
// Representing cards with bit fields in a struct
#include <stdio.h>
#define CARDS 52

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

typedef struct bitCard Card; // new type name for struct bitCard

void fillDeck(Card * const wDeck); // prototype
void deal(const Card * const wDeck); // prototype

int main(void)
{
        Card deck[CARDS]; // create array of Cards

        fillDeck(deck);

        puts("Card values 0-12 correspond to Ace through King");
        puts("Suit values 0-3 correspond Hearts, Diamonds, Clubs and Spades");
        puts("Color values 0-1 correspond to red and black\n");
        deal(deck);
}
```

Bit fields for efficient memory utilization

```
// initialize Cards
void fillDeck(Card * const wDeck)
{
	// loop through wDeck
	for (size_t i = 0; i < CARDS; ++i) {
		wDeck[i].face = i % (CARDS / 4);
		wDeck[i].suit = i / (CARDS / 4);
		wDeck[i].color = i / (CARDS / 2);
	}
}

// output cards in two-column format; cards 0-25 indexed with
// k1 (column 1); cards 26-51 indexed with k2 (column 2)
void deal(const Card * const wDeck)
{
	printf("%-6s%-6s%-15s%-6s%-6s%s\n", "Card", "Suit", "Color",
		"Card", "Suit", "Color");

	// loop through wDeck
	for (size_t k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; ++k1, ++k2) {
		printf("%-6d%-6d%-15d",
			wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
		printf("%-6d%-6d%d\n",
			wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
	}
}
```

Bit fields for efficient memory utilization

```
Card values 0-12 correspond to Ace through King
Suit values 0-3 correspond Hearts, Diamonds, Clubs and Spades
Color values 0-1 correspond to red and black

Card  Suit  Color        Card  Suit  Color
0     0     0            0     2     1
1     0     0            1     2     1
2     0     0            2     2     1
3     0     0            3     2     1
4     0     0            4     2     1
5     0     0            5     2     1
6     0     0            6     2     1
7     0     0            7     2     1
8     0     0            8     2     1
9     0     0            9     2     1
10    0     0            10    2     1
11    0     0            11    2     1
12    0     0            12    2     1
0     1     0            0     3     1
1     1     0            1     3     1
2     1     0            2     3     1
3     1     0            3     3     1
4     1     0            4     3     1
5     1     0            5     3     1
6     1     0            6     3     1
7     1     0            7     3     1
8     1     0            8     3     1
9     1     0            9     3     1
10    1     0            10    3     1
11    1     0            11    3     1
12    1     0            12    3     1
```