Programming Languages -1 (Introduction to C)

structures

Instructor: M.Fatih AMASYALI
E-mail:mfatih@ce.yildiz.edu.tr

Introduction

- Structures
  - A collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

<u>Example:</u>

```
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 *`**
    - These members are **`face`** and **`suit`**
  - Comparing structures is a syntax error.

Structure Definitions

<u>Example</u>:

A date consists of several parts, such as the day, month, and year, and the day of the year, and the month name

```
struct date {
    int day;
    int month;
    int year;
    int year_date;
    char month_name[4];
    };
```

- **`date`**: the name of the structure, called *structure tag*.
- **`day`**, **`month`**, …: the elements or variables mentioned in a structure are called *members*.

- **`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

- Declarations

  method 1: declared like other variables: declare tag first, and then declare variable.

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

  method 2: A list of variables can be declared after the right brace and use comma separated list:

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

```
struct date {
    .. .. ..
} d1, d2, d3;
struct date d4, d5;
```

- 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
- Initialization of Structures
  - Initializer lists

    Example:

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

    Example:

    ```
    struct date d1 = {4, 7, 1776, 186, “Jul”};
    struct date d2 = {4, 7, 1776, 186,
    {‘J’,’u’,’l’,’\0’}};
    ```

  - Assignment statements

    Example:

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

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

```
typedef struct card *CardPtr;
```

  - Defines a new type name **`CardPtr`** as a synonym for type **`struct card *`**
  - **`typedef`** does not create a new data type while it only creates an alias

Structures – initialize

You may initialize a variable corresponding to a structure that was defined by initializing all its elements as follows:

```
struct name var = {init_element_1, …, init_element_n}
```

```
#include <stdio.h>

struct address_struct
{ char *street;
  char *city_and_state;
  long zip_code;
};
typedef struct address_struct address;

void main()
{
  address a = { "1449 Crosby Drive", "Fort Washington, PA",
       19034 };
}
```

Initialization of structure array

```
struct person_data{
   char name[NAMESIZE];
   int tscore;
   int math;
   int english;
   }  person[]={
       {"Jane",180,89,91},
       {"John",190,90,100},
         .. .. .. ..
       };  /* similar to 2D array */
```

⇒

the inner brace is not necessary

```
"Jane",180,89,91,
"John",190,90,100,
.. .. .. ..
```

- Accessing structure members
  - Dot (**.**) is a member operator used with structure variables
    - Syntax: **`structure_name.member`**

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

    - One could also declare and initialize **`threeHearts`** as follows:

```
struct card threeHearts;
threeHearts.face = “Three”;
threeHearts.suit = “Hearts”;
```

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

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

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

If p is a pointer to a structure and x is an element of the structure then to access this element one puts: `(*p).x` or more commonly `p->x`

```
struct card_struct
{       int pips;
        char suit;
};
typedef struct card_struct card;

void set_card( card *c )
{       c->pips = 3;
        c->suit = 'A';
}
void main()
{       card a;
        set_card( &a );
}
```

```c
#include <stdio.h>

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

int main( void )
{
   struct card aCard; /* define one struct card variable */
   struct card *cardPtr; /* define a pointer to a struct card */

   /* place strings into aCard */
   aCard.face = "Ace";
   aCard.suit = "Spades";
```

Structure definition

Structure definition must end with semicolon

Dot operator accesses members of a structure

```c
   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 );

   return 0; /* indicates successful termination */

} /* end main */
```

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

Arrow operator accesses members of a structure pointer

- Structure can be nested

```
struct date {
    int day;
    int month;
    int year;
    int year_date;
    char month_name[4];
    };

struct person {
  char name [NAME_LEN];
  char address[ADDR_LEN};
  long zipcode;
  long ss__number;
  double salary;

  struct date birthday;
};

struct person emp;

emp.birthday.month = 6;
emp.birthday.year = 1776;
```

- Name Rule
  - Members in different structure can have the same name, since they are at different position.

```
struct s1 {
    .. .. .. ..
    char name[10];
    .. .. .. ..
    } d1;

struct s2 {
    .. .. .. ..
    int name;
    .. .. .. ..
    } d2;

struct s3 {
      .. .. .. ..
    int name;
    struct s2 t3;
    .. .. .. ..
    } d3;
float name;
```

Point in rectangular

```
struct point {int x,y;};
struct rect {struct point pt1, pt2;};

int ptinrect (struct point p, struct
  rect r)
{ return p.x>=r.pt1.x && p.x<r.pt2.x
      && p.y>=r.pt1.y && p.y<r.pt2.y
}
```

midpoint

```
struct point midpoint (struct point a,
  struct point b)
{
   struct m = {(a.x+b.x)/2, (a.y+b.y)/2};
   return m;
}
```

Card Shuffling and Dealing Simulation

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

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

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

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

int main( void )
{
   Card deck[ 52 ]; /* define array of Cards */

   /* initialize array of pointers */
   char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King"};

```

Each **`card`** has a face and a suit

`Card` is now an alias for **`struct card`**

```c
   /* initialize array of pointers */
   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 */

   return 0; /* indicates successful termination */

} /* end main */

/* place strings into Card structures */
void fillDeck( Card * wDeck, char * wFace[],
   char * wSuit[] )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = wFace[ i % 13 ];
      wDeck[ i ].suit = wSuit[ i / 13 ];
   } /* end for */

} /* end function fillDeck */

```

Fills the deck by giving each **Card** a face and suit

```c
/* shuffle cards */
void shuffle( Card * wDeck )
{
   int i;     /* counter */
   int j;     /* variable to hold random value between 0 - 51 */
   Card temp; /* define temporary structure for swapping Cards */

   /* loop through wDeck randomly swapping Cards */
   for ( i = 0; i <= 51; i++ ) {
      j = rand() % 52;
      temp = wDeck[ i ];
      wDeck[ i ] = wDeck[ j ];
      wDeck[ j ] = temp;
   } /* end for */

} /* end function shuffle */

/* deal cards */
void deal( Card * wDeck )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      printf( "%5s of %-8s%c", wDeck[ i ].face, wDeck[ i ].suit,
         ( i + 1 ) % 2 ? '\t' : '\n' );
   } /* end for */

} /* end function deal */
```

Each card is swapped with another, random card, shuffling the deck

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

Outline

Unions

- Similar to structures, but they can only hold one of the elements at a time
- So they use the same spot in memory to save any of the possible elements.
- Memory for union is max of memory needed for each element

```
union int_or_float
{
        int i;
        float f;
};

union int_or_float a;
```

```c
#include <stdio.h>

union number {
   int x;
   double y;
};

int main()
{
   union number value;

   value.x = 100;
   printf( "%s\n%s\n%s%d\n%s%f\n\n",
          "Put a value in the integer member",
          "and print both members.",
          "int:   ", value.x,
          "double:\n", value.y );

   value.y = 100.0;
   printf( "%s\n%s\n%s%d\n%s%f\n",
          "Put a value in the floating member",
          "and print both members.",
          "int:   ", value.x,
          "double:\n", value.y );
   return 0;
}
```

Define union

Initialize variables

Set variables

Print

Program Output

```
Put a value in the integer member
and print both members.
int:   100
double:
-
92559592117433136000000000000000000000000
0000000000000000.00000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

Memory Allocation Functions

All functions are declared in <stdlib.h>

- void *malloc(size) allocates size bytes and returns a pointer to the new space if possible; otherwise it returns the null pointer.
- void *calloc(n, size) is same as malloc(n*size), but the allocated storage is also zeroed (it is slower).
- void *realloc(void *ptr, size) changes size of previously allocated object to size and returns pointer to new space is possible (or NULL otherwise)
- void free(void *ptr) deallocates previously allocated storage

Some suggestions and comments

- The NULL pointer is a pointer that points to nothing. So if `p=NULL;` then the statement `*p` is going to produce a run-time error.
- void * is a generic pointer type that is returned by all memory functions.
- By generic one means that void * covers all possible pointers that exist and thus by declaring a pointer as generic (void *) you suggest that it can accommodate any possible pointer type!

Getting an array of numbers

```
#include <stdio.h>

void main()
{
  int *numbers, size, i, sum = 0;

  printf( "How many numbers? " );
  scanf( "%d", &size );
  numbers = malloc( size * sizeof( int ));
  for( i = 0; i < size; i++ )
    {
      scanf( "%d", &numbers[i] );
    }
  for( i = 0; i < size; i++ )
    {
      sum += numbers[i];
    }
  printf( "%d\n", sum );
  free( numbers );
}
```

- Enumeration
  - Set of integer constants represented by identifiers
  - Enumeration constants are like symbolic constants whose values are automatically set
    - Values start at **`0`** and are incremented by **`1`**
    - Values can be set explicitly with **`=`**
    - Need unique constant names

Example:

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

- Creates a new type enum Months in which the identifiers are set to the integers 1 to 12

```c
#include <stdio.h>

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

int main()
{
   enum months month;
   const char *monthName[] = { "", "January", "February",
                               "March", "April", "May",
                               "June", "July", "August",
                               "September", "October",
                               "November", "December" };

   for ( month = JAN; month <= DEC; month++ )
      printf( "%2d%11s\n", month, monthName[ month ] );

   return 0;
}
```

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

Bitwise Operators

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

Operator		Description
&	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1 .
|	bitwise inclusive OR	The bits in the result are set to 1 if at least one of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from 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.
~	one's complement	All 0 bits are set to 1 and all 1 bits are set to 0.

Bitwise operators.

```c
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main( void )
{
   unsigned x; /* variable to hold user input */

   printf( "Enter an unsigned integer: " );
   scanf( "%u", &x );

   displayBits( x );

   return 0; /* indicates successful termination */

} /* end main */

```

```
/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

   /* define displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%10u = ", value );

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

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

   } /* end for */

   putchar( '\n' );
} /* end function displayBits */
```

`displayMask` is a 1 followed by 31 zeros

Bitwise AND returns nonzero if the leftmost bits of **`displayMask`** and **`value`** are both 1, since all other bits in **`displayMask`** are 0s.

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

```c
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

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

   /* demonstrate bitwise AND (&) */
   number1 = 65535;
   mask = 1;
   printf( "The result of combining the following\n" );
   displayBits( number1 );
   displayBits( mask );
   printf( "using the bitwise AND operator & is\n" );
   displayBits( number1 & mask );

```

Bitwise AND sets each bit in the result to 1 if the corresponding bits in the operands are both 1

```
   /* demonstrate bitwise inclusive OR (|) */
   number1 = 15;
   setBits = 241;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( setBits );
   printf( "using the bitwise inclusive OR operator | is\n" );
   displayBits( number1 | setBits );

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

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

   return 0; /* indicates successful termination */
} /* end main */
```

Bitwise inclusive OR sets each bit in the result to 1 if at least one of the corresponding bits in the operands is 1

Bitwise exclusive OR sets each bit in the result to 1 if only one of the corresponding bits in the operands is 1

Complement operator sets each bit in the result to 0 if the corresponding bit in the operand is 1 and vice versa

```c
/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

   /* declare displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%10u = ", value );

   /* loop through bits */
   for ( 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( ' ' );
      } /* end if */

   } /* end for */

   putchar( '\n' );
} /* end function displayBits */
```

```
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
```

Common Programming Error

- Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is an error.

Bit fields

- 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. Bit fields are not "arrays of 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;
};
```

```c

#include <stdio.h>

/* 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 */

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

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

int main( void )
{
   Card deck[ 52 ]; /* create array of Cards */

   fillDeck( deck );
   deal( deck );

   return 0; /* indicates successful termination */

} /* end main */

```

Bit fields determine how much memory each member of a structure can take up

```
/* initialize Cards */
void fillDeck( Card * wDeck )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = i % 13;
      wDeck[ i ].suit = i / 13;
      wDeck[ i ].color = i / 26;
   } /* end for */

} /* end function fillDeck */

/* output cards in two column format; cards 0-25 subscripted with
   k1 (column 1); cards 26-51 subscripted k2 (column 2) */
void deal( Card * wDeck )
{
   int k1; /* subscripts 0-25 */
   int k2; /* subscripts 26-51 */

   /* loop through wDeck */
   for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      printf( "Card:%3d  Suit:%2d  Color:%2d   ",
         wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
      printf( "Card:%3d  Suit:%2d  Color:%2d\n",
         wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
   } /* end for */

} /* end function deal */
```

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

Self-referential structures

- A structure that has as its element(s) pointers to the structure itself is called a self-referential structure. E.g. a tree:

```
typedef struct tree {
   DATA d;
   struct tree *left, *right;
};
```

- Other classical structures include: stacks, queues and linked lists…

Linear Linked Lists

- Problem with arrays: If we assume that they're of fixed length, need to know maximum length ahead of time. Unrealistic.
- Also, even if we did know the maximum length ahead of time, if array was not full, we would be "wasting memory."
- One solution: Linear linked lists.

```
typedef char DATA;

struct linked_list
{
      DATA d;
      struct linked_list *next;    /* refers to self */
};

typedef struct linked_list ELEMENT;
typedef ELEMENT *LINK;
```

Linear Linked Lists: List creation

```
/* Uses recursion.  From Kelley/Pohl. */

LINK string_to_list( char s[] )
{
  LINK head;

  if( s[0] == '\0' )
    return NULL;
  else
  {
    head = (LINK) malloc( sizeof( ELEMENT ));
    head->d = s[0];
    head->next = string_to_list( s + 1 );
    return head;
  }
}
```

Linear Linked Lists: Counting

```
int count( LINK head )  /* recursively: Kelley/Pohl */
{
  if( head == NULL )
    return 0;
  else
    return( 1 + count( head->next ));
}

int count_it( LINK head )
{
  int cnt;
  for( cnt = 0; head != NULL; head = head->next )
  {
    cnt++;
  }
  return cnt;
}
```

Linear Linked Lists: Lookup

```
/* from Kelley/Pohl */

LINK lookup( DATA c, LINK head )
{
  if( head == NULL )
    return NULL;
  else if( c == head->d )
    return head;
  else
    return( lookup( c, head->next ));
}
```

Linear Linked Lists: Insertion/Deletion

```
/* from Kelley/Pohl */

/* Assumes q is a one-element list, and inserts it between
   p1 and p2, assumed to be consecutive cells in some list */
void insert( LINK p1, LINK p2, LINK q )
{
  p1->next = q;
  q->next = p2;
}

void delete_list( LINK head )
{
  if( head != NULL )
  {
    delete_list( head->next );
    free( head );
  }
}
```

Linear Linked Lists: write list, main

```
void writelist( LINK head )
{
  while(head != NULL)
  {
    putchar(head->d);
    head = head->next;
  }
  printf("\n");
}
int main()
{
   LINK liste,liste2,liste3,liste4;
   liste=string_to_list("abc123");
   printf("%d %d \n",count(liste),count_it(liste));
   liste2=lookup( 'c', liste );
   writelist(liste2);
   liste3=string_to_list("efgh");
   liste4=string_to_list("d");
   insert(liste2,liste3,liste4);
   writelist(liste2);    writelist(liste);
   getch(); return 0;
}
```

Output:
6 6
c123
cdefgh
abcdefgh

Stacks

- Another form of data abstraction: will implement using ideas similar to those used in implementing linear linked list.
- Only two basic operations defined on a stack: push (insertion), pop (deletion).
- Access is restricted to the “head” of the stack.

```
#define NULL 0
typedef char DATA;

struct stack
{
      DATA d;
      struct stack *next;     /* refers to self */
};

typedef struct stack ELEMENT;
typedef ELEMENT *LINK;
```

Stacks: Testing for emptiness, Top element

```
int isempty( TOP t )
{
  return( t == NULL );
}

DATA vtop( TOP t )
{
  return( t -> d );
}
```

Stacks: Pop

```
/* remove top element from the stack */
void pop( TOP *t, DATA *x )
{
  TOP t1 = *t;

  if( !isempty( t1 ))
  {
    *x = t1->d;
    *t = t1->next;
    free( t1 );
  }
  else
    printf( “Empty stack.\n” );
}
```

Stacks: Push

```
/* put an element at the top of the stack */
void push( TOP *t, DATA x )
{
  TOP temp;

  temp = malloc( sizeof( ELEMENT ));
  temp->d = x;
  temp->next = *t;
  *t = temp;
}
```

Referance

- Ioannis A. Vetsikas, Lecture notes
- Dale Roberts, Lecture notes