EECS 10: Computational Methods in Electrical and Computer Engineering

Lecture 9

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 9.1: Overview

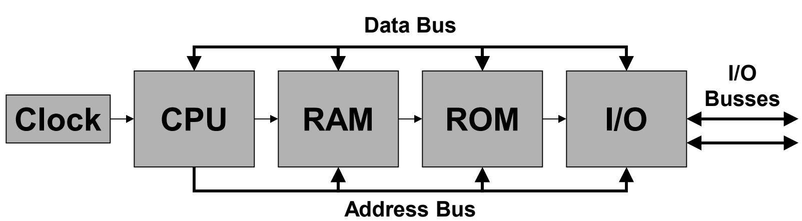
- Course Administration
  - Final course evaluation
- Basic Computer Architecture
  - Computer components
- Binary Data Representation
  - Bits, bytes, and words
  - Memory sizes
  - Memory format
  - Number systems
  - Memory segmentation

Course Administration

- Final Course Evaluation
  - Open this week
  - July 22, 2017, through Sunday, July 30, 2017
  - Online via EEE Evaluation application
- Mandatory Evaluation of Course and Instructor
  - Voluntary
  - Anonymous
  - Very valuable
    - Help to improve this class!
- Please spend 5 minutes!

Basic Computer Architecture

- Essential Computer Components
  - Central Processing Unit (CPU)
    - e.g. Intel Pentium, Motorola PowerPC, Sun SPARC, ...
  - Random Access Memory (RAM)
    - storage for program and data, read and write access
  - Read Only Memory (ROM)
    - fixed storage for basic input/output system (BIOS)
  - I/O Units
    - Input/output interfaces connecting to peripherals



Binary Data Representation

- Data and instructions in a computer are represented in binary format
  - 1 *bit* (binary digit), 2 possible values ■ □
    - 0 (false, "no", power off, "empty", ...)
    - 1 (true, "yes", power on, "filled", ...)
  - 1 *byte* = 8 bits (2^8 = 256 values) ■□■□□□■□
    - in C, type **`char`** equals one byte*
  - 1 *word* = 4 bytes* (2^{32} = 4294967296 values)
    - in C, type **`int`** equals one word

■□■□□□■■
□■■□■□■□
■□■□□■■□
■□■■□□■■

- Memory size is measured in Bytes
  - 1 KB = 1024 byte = 1 "kilo byte"
  - 1 MB = 1024*1024 byte = 1 "mega byte"
  - 1 GB = 1024*1024*1024 byte = 1 "giga byte"
  - 1 TB = 1024^4 byte = 1 "tera byte"

(*architecture dependent!)

EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 5

Binary Data Representation

- Memory is composed of addressable bytes
  - Example:
    1 KB of memory
  - What is the value at address 7?

```
7  □■□□■■□■
    7 6 5 4 3 2 1 0

= 0*2^7 + 1*2^6 + 0*2^5 + 0*2^4
+ 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0

= 0*128+ 1*64 + 0*32 + 0*16
+ 1*8  + 1*4  + 0*2  + 1*1

= 64 + 8 + 4 + 1

= 77
```

0	■□■□□□■■
1	□■■□■□■□
2	■□■□□■■□
3	□□□■□□■■
4	■□■■■□■□
5	□■□□■□■□
6	■□□□□■■□
7	□■□□■■□■
8	■□■□□□■□
9	□■□□■□■□
10	■□■□□■□□
11	□□■■□□■■
	. . .
1020	■□□□■□■■
1021	□■■□■□■□
1022	■□□□□□■□
1023	■□□■■□■■

EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 6

Binary Data Representation

- Review: Number Systems
  - DEC: Decimal numbers
    - Base 10, digits 0, 1, 2, 3, ..., 9
    - e.g. $157 = 1*10^2 + 5*10^1 + 7*10^0$
  - BIN: Binary numbers
    - Base 2, digits 0, 1
    - e.g. $10011101_2 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + ... + 1*2^0$
  - OCT: Octal numbers
    - Base 8, digits 0, 1, 2, 3, ..., 7
    - e.g. $235_8 = 2*8^2 + 3*8^1 + 5*8^0$
  - HEX: Hexadecimal numbers
    - Base 16, digits 0, 1, 2, 3, ..., 9, A, B, C, ..., F
    - e.g. $9D_{16} = 9*16^1 + 13*16^0$

Binary Data Representation

- Review: Number Systems

DEC	BIN	OCT	HEX
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Binary Data Representation

- Review: Number Systems (signed/unsigned)

SDEC	UDEC	BIN	OCT	HEX
0	0	0000	0	0
1	1	0001	1	1
2	2	0010	2	2
3	3	0011	3	3
4	4	0100	4	4
5	5	0101	5	5
6	6	0110	6	6
7	7	0111	7	7
-8	8	1000	10	8
-7	9	1001	11	9
-6	10	1010	12	A
-5	11	1011	13	B
-4	12	1100	14	C
-3	13	1101	15	D
-2	14	1110	16	E
-1	15	1111	17	F

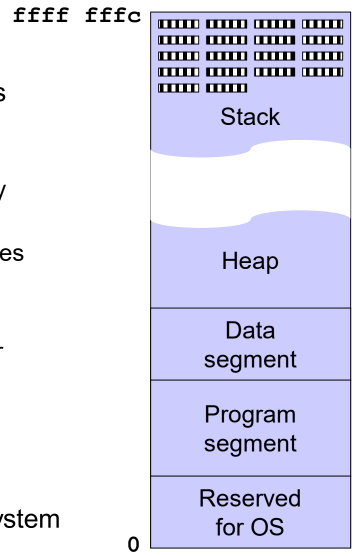
Binary Data Representation

- Review: Number Systems
  - Signed representation: *two's complement*
    - to obtain the negative of any number in binary representation, ...
      - ... invert all bits,
      - ... and add 1
  - Example: 4-bit two's complement

SDEC	UDEC	BIN	OCT	HEX
...	...	...	...	...
7	7	0111	7	7
-8	8	1000	10	8
-7	9	1001	11	9
...	...	...	...	...

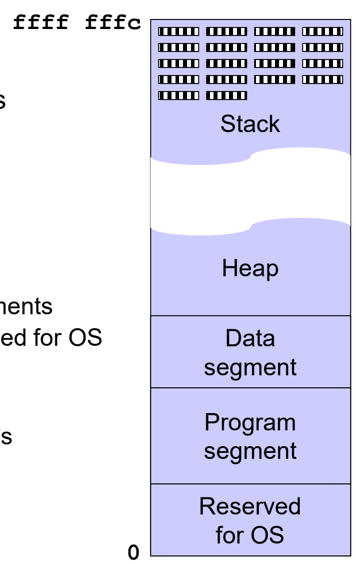
Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 4 GB of memory
  - Stack
    - grows and shrinks dynamically
    - function call hierarchy
    - stack frames with local variables
  - Heap
    - “free” storage
    - dynamic allocation by the user
  - Data segment
    - global (and static) variables
  - Program segment
    - stores binary program code
  - Reserved area for operating system



Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 4 GB of memory
- Memory errors
  - *Out of memory*
    - Stack and heap collide
  - *Segmentation fault*
    - access outside allocated segments
    - e.g. access to segment reserved for OS
  - *Bus error*
    - mis-aligned word access
    - e.g. word access to an address that is not divisible by 4



Lecture 9.2: Overview

- Data Structures
  - Objects in memory
  - Pointers
    - Pointer definition
    - Pointer initialization, assignment
    - Pointer dereferencing
  - Pointer arithmetic
    - Increment, decrement
  - Pointer comparison

Objects in Memory

- Data in memory is organized as a set of objects
- Every object has ...
  - ... a *type* (e.g. **`int`**, **`double`**, **`char[5]`**)
    - type is known to the compiler at compile time
  - ... a *value* (e.g. **`42`**, **`3.1415`**, **`"text"`**)
    - value is used for computation of expressions
  - ... a *size* (number of bytes in the memory)
    - in C, the **`sizeof`** operator returns the size of a variable or type
  - ... a *location* (address in the memory)
    - in C, the "address-of" operator (**`&`**) returns the address of an object
- Variables ...
  - ... serve as identifiers for objects
  - ... are bound to objects
  - ... give objects a name

Objects in Memory

- Example: Variable values, addresses, and sizes

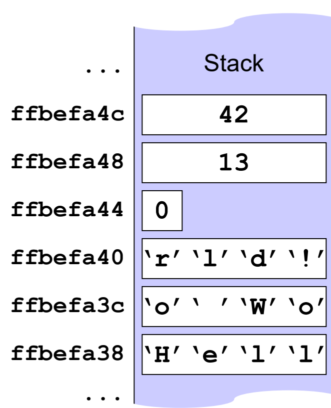
```
int x = 42;
int y = 13;
char s[] = "Hello World!";

printf("Value   of x    is %d.\n", x);
printf("Address of x    is %p.\n", &x);
printf("Size    of x    is %u.\n", sizeof(x));
printf("Value   of y    is %d.\n", y);
printf("Address of y    is %p.\n", &y);
printf("Size    of y    is %u.\n", sizeof(y));
printf("Value   of s    is %s.\n", s);
printf("Address of s    is %p.\n", &s);
printf("Size    of s    is %u.\n", sizeof(s));
printf("Value   of s[1] is %c.\n", s[1]);
printf("Address of s[1] is %p.\n", &s[1]);
printf("Size    of s[1] is %u.\n", sizeof(s[1]));
```


Objects in Memory

- Example: Variable values, addresses, and sizes

```
int x = 42;
int y = 13;
char s[] = "Hello World!";
...
```

```
Value   of x    is 42.
Address of x    is ffbefa4c.
Size    of x    is 4.
Value   of y    is 13.
Address of y    is ffbefa48.
Size    of y    is 4.
Value   of s    is Hello World!.
Address of s    is ffbefa38.
Size    of s    is 13.
Value   of s[1] is e.
Address of s[1] is ffbefa39.
Size    of s[1] is 1.
```



Pointers

- *Pointers* are variables whose values are *addresses*
  - The *“address-of”* operator (`&`) returns a pointer!
- Pointer Definition
  - The unary `*` operator indicates a pointer type in a definition

```
int x = 42;           /* regular integer variable */
int *p;               /* pointer to an integer */
```

- Pointer initialization or assignment
  - A pointer may be set to the “address-of” another variable

```
p = &x;               /* p points to x */
```

  - A pointer may be set to `0` (points to no object)

```
p = 0;                /* p points to no object */
```

  - A pointer may be set to `NULL` (points to “NULL” object)

```
#include <stdio.h>    /* defines NULL as 0 */
p = NULL;             /* p points to no object */
```


Pointers

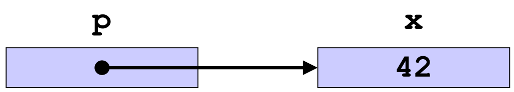
- Pointer Dereferencing
  - The unary `*` operator dereferences a pointer to the value it points to (*“content-of”* operator)

```
#include <stdio.h>

int  x = 42;    /* regular integer variable */
int *p = NULL; /* pointer to an integer */
```

Pointers

- Pointer Dereferencing
  - The unary `*` operator dereferences a pointer to the value it points to (“content-of” operator)

```
#include <stdio.h>

int  x = 42;    /* regular integer variable */
int *p = NULL;  /* pointer to an integer */

p = &x;         /* make p point to x */
```



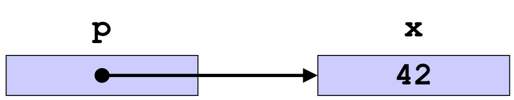
EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 19

Pointers

- Pointer Dereferencing
  - The unary `*` operator dereferences a pointer to the value it points to (“content-of” operator)

```
#include <stdio.h>

int  x = 42;    /* regular integer variable */
int *p = NULL;  /* pointer to an integer */

p = &x;         /* make p point to x */
printf(“x is %d, content of p is %d\n”, x, *p);
```

```
x is 42, content of p is 42
```



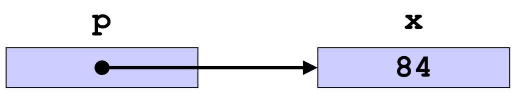
EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 20

Pointers

- Pointer Dereferencing
  - The unary ***** operator dereferences a pointer to the value it points to ("content-of" operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */

p = &x;        /* make p point to x */
printf("x is %d, content of p is %d\n", x, *p);
*p = 2 * *p;   /* multiply content of p by 2 */
printf("x is %d, content of p is %d\n", x, *p);
```

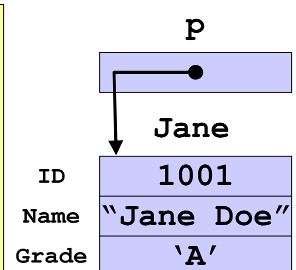
```
x is 42, content of p is 42
x is 84, content of p is 84
```



Pointers

- Pointer Dereferencing
  - The **->** operator dereferences a pointer to a structure to the content of a structure member

```
struct Student
{  int  ID;
   char Name[40];
   char Grade;
};

struct Student Jane =
{1001, "Jane Doe", 'A'};

struct Student *p = &Jane;

void PrintStudent(void)
{
   printf("ID:    %d\n", p->ID);
   printf("Name:  %s\n", p->Name);
   printf("Grade: %c\n", p->Grade);
}
```




```
ID:    1001
Name:  Jane Doe
Grade: A
```

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
```

```
20,
```


Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30,
```

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
p--;                         /* decrement p by 1 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30, 20,
```


Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
p--;                         /* decrement p by 1 */
printf("%d, ", *p);          /* print content of p */
p += 2;                      /* increment p by 2 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30, 20, 40,
```

Pointers

- Pointer Comparison
  - Pointers may be compared for equality
    - operators **==** and **!=** are useful to determine *identity*
    - operators **<**, **<=**, **>=**, and **>** are *not* applicable

```
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;                /* pointers to integer */
p1 = &x[1]; p2 = &x[3];      /* point to x[1], x[3] */

if (p1 == p2)
  { printf("p1 and p2 are identical!\n");
   }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are the same!\n");
   }
```

```
Contents of p1 and p2 are the same!
```


Pointers

- Pointer Comparison
  - Pointers may be compared for equality
    - operators **==** and **!=** are useful to determine *identity*
    - operators **<**, **<=**, **>=**, and **>** are *not* applicable

```
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;                /* pointers to integer */
p1 = &x[1]; p2 = &x[3];      /* point to x[1], x[3] */
p1 += 2;                     /* increment p1 by 2 */
if (p1 == p2)
  { printf("p1 and p2 are identical!\n");
   }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are the same!\n");
   }
```

```
p1 and p2 are identical!
Contents of p1 and p2 are the same!
```

Lecture 9.3: Overview

- Pointers
  - String operations using pointers
    - Pointer and array type equivalence
    - Passing pointers to functions
    - Type qualifier **`const`**
  - Standard library functions
    - String operations defined in **`string.h`**
  - Example
    - **`Bubblesort2.c`**

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
   int l = 0;
   char *p = s;

   while(*p != 0)
   { p++;
     l++;
   }
   return l;
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(&s1[0]));
printf("Length of %s is %d\n",
            s2, Length(&s2[0]));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
    int l = 0;
    char *p = s;

    while(*p != 0)
    { p++;
      l++;
    }
    return l;
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(&s1[0]));
printf("Length of %s is %d\n",
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s2** is an array, but can be passed as a pointer argument
    - Character array **s2** is same as character pointer **&s2[0]**

EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 31

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char *s)
{
    int l = 0;
    char *p = s;

    while(*p != 0)
    { p++;
      l++;
    }
    return l;
}
```

```
char s1[] = "ABC";
char *s2  = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(s1));
printf("Length of %s is %d\n",
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s1** is an array of characters, **s2** is a pointer to character
    - Both **s1** and **s2** can be passed to character pointer **s**

EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 32

Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char s[])
{
    int l = 0;
    char *p = s;

    while(*p != 0)
    { p++;
      l++;
    }
    return l;
}
```

```
char s1[] = "ABC";
char *s2  = "Hello World!";

printf("Length of %s is %d\n",
            s1, Length(s1));
printf("Length of %s is %d\n",
            s2, Length(s2));
```

```
Length of ABC is 3
Length of Hello World! is 12
```

  - Array and pointer types are equivalent
    - **s1** is an array of characters, **s2** is a pointer to character
    - Both **s1** and **s2** can be passed to character array **s**

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
     char *Dst,
     char *Src)
{
 do{
     *Dst = *Src;
     Dst++;
   } while(*Src++);
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Passing pointers = Pass by reference!

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
        char *Dst,
      const char *Src)
{
 do{
        *Dst = *Src;
        Dst++;
    } while(*Src++);
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Type qualifier `const`:
      Modification by pointer derefencing *not* allowed!

EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 35

Pointers

- String Operations using Pointers
  - Example: String copy

```
void Copy(
      const char *Dst,
      const char *Src)
{
 do{
        *Dst = *Src;
        Dst++;
    } while(*Src++);
}
```

Error!
Write access to
`const` data!

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n",
               s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n",
               s1, s2);
```

```
s1 is ABC, s2 is Hello World!
s1 is ABC, s2 is ABC
```

  - Passing pointers as arguments to functions
    - Function can modify caller data by pointer dereferencing
    - Type qualifier `const`:
      Modification by pointer derefencing *not* allowed!

EECS10: Computational Methods in ECE, Lecture 9 (c) 2017 R. Doemer 36

Standard Library Functions

- Functions declared in **string.h** (part 1/2)
  - **typedef unsigned int size_t;**
    - type definition for length of strings
  - **size_t strlen(const char *s);**
    - returns the length of string **s**
  - **int strcmp(const char *s1, const char *s2);**
    - alphabetically compares string **s1** with string **s2**
    - returns -1 / 0 / 1 for less-than / equal-to / greater-than
  - **int strncmp(const char *s1, const char *s2, size_t n);**
    - same as previous, but compares maximal **n** characters
  - **int strcasecmp(const char *s1, const char *s2);**
  - **int strncasecmp(const char *s1, const char *s2, size_t n);**
    - same as string comparisons above, but case-insensitive

Standard Library Functions

- Functions declared in **string.h** (part 2/2)
  - **char *strcpy(char *s1, const char *s2);**
    - copies string **s2** into string **s1**
  - **char *strncpy(char *s1, const char *s2, size_t n);**
    - copies maximal **n** characters of string **s2** into string **s1**
  - **char *strcat(char *s1, const char *s2);**
    - concatenates string **s2** to string **s1**
  - **char *strncat(char *s1, const char *s2, size_t n);**
    - concatenates maximal **n** characters of string **s2** to string **s1**
  - **char *strchr(const char *s, int c);**
    - returns a pointer to the first character **c** in string **s**, or **NULL** if not found
  - **char *strrchr(const char *s, int c);**
    - returns a pointer to the last character **c** in string **s**, or **NULL** if not found
  - **char *strstr(const char *s1, const char *s2);**
    - returns a pointer to the first appearance of **s2** in string **s1** (or **NULL**)

Pointers

- Case Study Revisited: *Bubble Sort*
  - Task: Sort an array of strings alphabetically
  - Input: Array of 10 strings entered by the user
  - Output: Array of 10 strings in alphabetical order
- Approach: Divide and Conquer
  - Step 1: Let user enter 10 strings
  - Step 2: Sort the array of strings
    - Algorithm
      - in 9 rounds, compare all adjacent pairs of strings and swap the pair if they are not in alphabetical order
    - String comparison
      - ➢ use standard library function `strcmp()`
    - String swap (exchange two strings)
      - ➢ swap pointers to the two strings (higher efficiency!)
  - Step 3: Output the strings in order

Pointers

- Program example: **`BubbleSort2.c`** (part 1/6)

```
/* BubbleSort.c: sort strings alphabetically    */
/* author: Rainer Doemer                        */
/* modifications:                               */
/* 09/02/13 RD  pointer table for efficiency    */
/* 11/01/06 RD  swap only adjacent elements     */
/* 11/06/04 RD  initial version                 */

#include <stdio.h>
#include <string.h>

/* constant definitions */
#define NUM 10  /* ten strings  */
#define LEN 20  /* of length 20 */

/* function declarations */
void EnterText(char Text[NUM][LEN], char *P[NUM]);
void PrintText(char *P[NUM]);
void SwapStrings(char *P[NUM], int i, int j);
void BubbleSort(char *P[NUM]);
...
```

Pointers

- Program example: **BubbleSort2.c** (part 2/6)

```
...

/* function definitions */

/* let the user enter the text array          */

void EnterText(char Text[NUM][LEN], char *P[NUM])
{
   int i;

   for(i = 0; i < NUM; i++)
      { printf("Enter text string %2d: ", i+1);
        scanf("%19s", Text[i]);
        P[i] = Text[i];
       } /* rof */
} /* end of EnterText */

...
```


Pointers

- Program example: **BubbleSort2.c** (part 3/6)

```
...

/* print the text array on the screen         */

void PrintText(char *P[NUM])
{
   int i;

   for(i = 0; i < NUM; i++)
      { printf("String %2d: %s\n", i+1, P[i]);
       } /* rof */
} /* end of PrintText */

...
```

Pointers

- Program example: **BubbleSort2.c** (part 4/6)

```
...

/* swap/exchange the pointers to two strings */

void SwapStrings(char *P[NUM], int i, int j)
{
    char *tmp;

    tmp = P[i];
    P[i] = P[j];
    P[j] = tmp;

} /* end of SwapStrings */

...
```


Pointers

- Program example: **BubbleSort2.c** (part 5/6)

```
...
/* sort the text array by comparing every pair  */
/* of strings; if the pair of strings is not in */
/* alphabetical order, swap it                  */

void BubbleSort(char *P[NUM])
{
   int p, i;

   for(p = 1; p < NUM; p++)
      { for(i = 0; i < NUM-1; i++)
           { if (strcmp(P[i], P[i+1]) > 0)
                { SwapStrings(P, i, i+1);
                 } /* fi */
            } /* rof */
       } /* rof */
} /* end of BubbleSort */

...
```

Pointers

- Program example: **`BubbleSort2.c`** (part 6/6)

```
...
/* main function: enter, sort, print the text    */

int main(void)
{   /* local variables */
    char Text[NUM][LEN]; /* NUM strings, length LEN */
    char *P[NUM];         /* NUM pointers to strings */

    /* input section */
    EnterText(Text, P);

    /* computation section */
    BubbleSort(P);

    /* output section */
    PrintText(P);

    /* exit */
    return 0;
} /* end of main */

/* EOF */
```


Pointers

- Example session: **`BubbleSort2.c`**

```
% vi BubbleSort2.c
% gcc BubbleSort2.c -o BubbleSort2 -Wall -ansi
% BubbleSort2
Enter text string  1: Sun
Enter text string  2: Mercury
Enter text string  3: Venus
Enter text string  4: Earth
Enter text string  5: Mars
Enter text string  6: Jupiter
Enter text string  7: Saturn
Enter text string  8: Uranus
Enter text string  9: Neptune
Enter text string 10: Pluto
String  1: Earth
String  2: Jupiter
String  3: Mars
String  4: Mercury
String  5: Neptune
String  6: Pluto
String  7: Saturn
String  8: Sun
String  9: Uranus
String 10: Venus
%
```