Microprocessor Programming

Dr. Tin Ni Ni Kyaw

Ph. D (Kumamoto University, Japan)

Associate Professor

Department of Computer Engineering and Information Technology

Yangon Technological University

Yangon, Myanmar

Microprocessor Programming

Lecture 11

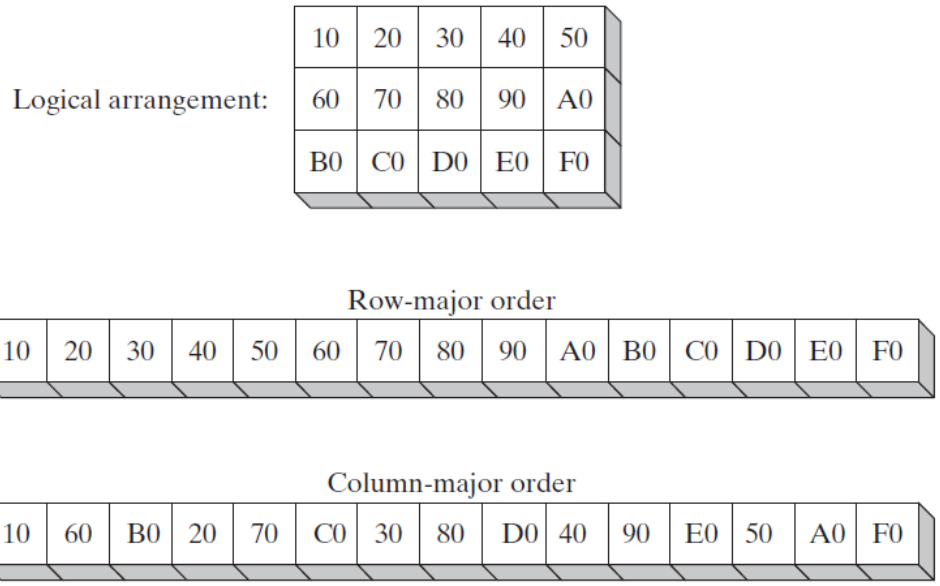
Arrays

Contents

- Introduction
- Arrays
- Indexing Using Registers
- Sequential Search
- Complete Programs
- Summary
- Practical Works
- Assignments

Introduction

- From an assembly language programmer's perspective, a two-dimensional array is a high-level abstraction of a one-dimensional array.
- There exist two methods of arranging rows and columns in memory: row-major and column-major orders as shown in figure.



Array Declaration

- There are a couple of different ways to declare an array based on the data and the needs of the programmer.
- The simplest way to declare an array is to list memory location after memory location.
- Note that number is declared as an sdword which takes up 4 bytes.

```
number  sdword 2
result      sdword 7
```

Array Declaration (Cont.)

- It is also possible to create an array as follow.
- It should be noticed that the subsequent memory locations do not have variable names attached to them.

```
numary  sdword 2
        sdword 5
        sdword 7
```

Array Declaration (Cont.)

- MASM has an easier way to declare the above on just one line.
- The directive sdword need to appear only once.
- Each of the entries would appear on the same line and each separated by a comma as follow.

```
numary  sdword 2,5,7
```

Array Initialization

- If each element of an array can be initialized to the same number, such as 0, it can be declared as follow:

```
zeroary   sdword 0,0,0
```

- If each memory location in the array did not need to be initialized, it can be declared as follow:

```
empary  sdword ?,?,?
```

Array Initialization (Cont.)

DUP Operator

- If there were hundreds of elements needed to be initialized, then clearly, the above method would be cumbersome.
- Instead, the DUP operator is very convenient.
- The above would be rewritten as follows:

```
zeroary   sdword 3 dup(0)
empary    sdword 3 dup(?)
```

Accessing Array Elements

- Assume that the last element of the previously declared numary needed to be moved to the first element of numary.
- Arrays always start with the zeroth element.
- The C equivalent of this operation would be:

```
numary[0] = numary[2];
```

Accessing Array Elements (Cont.)

- The equivalent assembly code would be as follow.

```
mov eax,numary+8        ; load eax with third element
mov numary+0,eax        ; store eax in first element
```

- Here, 8 is added to numary because the third element is in the thirdth memory location and each memory location is 4 bytes long.

Indexing Using Base Register

- In assembly language, indexing is accomplished using registers.
- The ebx register is known as the base register and it is very useful when indexing arrays.
- Although it is a register, it is used much like an index variable in the C programming language.

Indexing Using Base Register (Cont.)

- Let's consider a MASM code segment to calculate the sum of all the elements of an array.
- Assume that the array called numary already contains values and the equivalent C code is as shown below:

```
int numary[3] = {2,5,7};
sum = 0;
for(i=0; i<3; i++)
sum += numary[i];
```

- In the above C code, the variable i is being used both as a loop counter and as an index.

Indexing Using Base Register (Cont.)

- In assembly language, two separate registers need to be used for a loop counter and an index.
- So, ebx is used for indexing and ecx is used for loop control.
- Note that a 4 needs to be added to the ebx register to access the next signed double word.

Indexing Using Base Register (Cont.)

- The resulting code is as follows:

```
numary  sdword 2,5,7
sum     sdword ?
        mov sum,0                   ; initialize sum to 0
        mov ecx,3                   ; initialize ecx to 3
        mov ebx,0                   ; initialize ebx to 0
        .repeat
        mov eax,numary[ebx]         ; load eax with element of numary
        add sum,eax                 ; add eax to sum
        add ebx,4                   ; increment ebx by 4
        .untilcxz
```

Indexing Using Base Register (Cont.)

Example

- Write a MASM program for the following C program.

```
int arry[20],n,i;
printf("\n%s","Enter the number of integers to be input: ");
scanf("%d",&n);
if (n>0){
   for (i=0; i<n; i++){
      printf("\n%s","Enter an integer: ");
      scanf("%d",&arry[i]);
   }
   printf("\n%s\n\n","Reversed");
   for (i=n-1;i>=0;i--)
      printf("    %d\n\n",arry[i]);
}
else
   printf("\n%s\n\n","No data entered.");
```

Indexing Using Base Register (Cont.)

Example

- The following is a MASM code segment for the given C program.

```
            .data
msg1fmt     byte 0Ah,"%s",0
msg2fmt     byte 0Ah,"%s",0Ah,0Ah,0
msg3fmt     byte " %d",0Ah,0Ah, 0
in1fmt      byte "%d",0
msg1        byte "Enter the number of integers to be input: ",0
msg2        byte "Enter an integer: ",0
msg3        byte "Reversed",0
msg4        byte "No data entered."
n           sdword ?
arry        sdword 20 dup(?)
            .code
            INVOKE printf, ADDR msg1fmt,ADDR msg1
            INVOKE scanf, ADDR in1fmt, ADDR n
            mov ecx,n                       ; initialize ecx to n
            mov ebx,0                       ; initialize ebx to 0
            .if ecx>0
            .repeat
            push ecx                        ; save ecx
            INVOKE printf, ADDR msg1fmt, ADDR msg2
            INVOKE scanf, ADDR in1fmt, ADDR arry[ebx]
            pop ecx                         ; restore ecx
            add ebx,4                       ; increment ebx by 4
            .untilcxz
            INVOKE printf, ADDR msg2fmt, ADDR msg3
            mov ecx,n                       ; initialize ecx to n
            sub ebx,4                       ; subtract 4 from ebx
            .repeat
            push ecx                        ; save ecx
            INVOKE printf, ADDR msg3fmt,arry[ebx]
            pop ecx                         ; restore ecx
            sub ebx,4                       ; decrement ebx by 4
            .untilcxz
            .else
            INVOKE printf, ADDR msg2fmt, ADDR msg4
            .endif
```

Indexing Using esi and edi Registers

- There exist two index registers called esi (source index register) and edi (destination index register).
- These registers are used more like pointers and they are very useful when manipulating strings.
- As a general rule, when retrieving data from memory, it is best to use the esi register because memory is the source from which the data is coming.
- When storing data back into memory, the edi register indicates the destination where the data will be placed.

Indexing Using esi and edi Registers (Cont.)

OFFSET Operator

- It is important to load the addresses when working with the arrays.
- This can be done by using the OFFSET operator.
- In a sense, using OFFSET is static and the address is calculated at assembly time.
- To load the address of one element of an array into one register, we can use the MOV instruction together with the OFFSET operator.

Indexing Using esi and edi Registers (Cont.)

OFFSET Operator

- The following code segment loads the address of the second element of the array numary into the esi register by using MOV and OFFSET operator.

```
mov esi,offset numary+4
mov eax,[esi]
```

- If the square bracket around esi is omitted, then the value in the esi register will be transferred to the eax register.

Indexing Using esi and edi Registers (Cont.)

LEA Operator

- LEA stands for Load Effective Address.
- Using LEA is dynamic.
- With LEA, the address is calculated at run-time.
- Instead of using the MOV instruction with the word OFFSET, the LEA instruction can be used as follow.

```
lea esi,memory+4
mov eax,[esi]
```

Indexing Using esi and edi Register (Cont.)

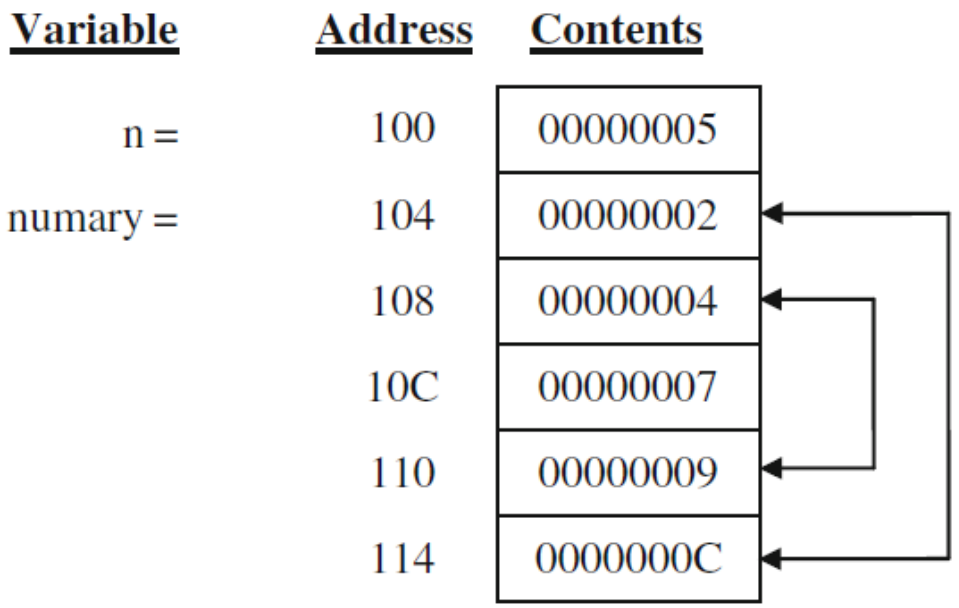
- The MASM code segment for calculating the sum of all the elements in an array can be implemented using the esi register as follows:

```
mov sum,0          ; initialize sum to zero
mov ecx,3          ; initialize ecx to 3
lea esi,numary+0 ; load the address of numary into esi
.repeat
mov eax,[esi]      ; move contents of where esi pointing to eax
add sum,eax        ; add eax to sum
add esi,4          ; increment esi by 4 to next element
.untilcxz
```

Indexing Using esi and edi Register (Cont.)

- Again, consider the MASM code segment to reverse the contents of the given array using the esi and edi registers.

```
n         sdword 5
numary    sdword 2,4,7,9,12
```



Indexing Using esi and edi Register (Cont.)

- Since there are an odd number of items, the middle element does not need to be swapped.
- Integer division can then be used to determine the number of items that need to be swapped.
- In this case, 5 divided by 2 is 2 and the two elements need to be exchanged.
- So, the loop should iterate a fixed number of times as n/2.

Indexing Using esi and edi Register (Cont.)

- The esi register is loaded with the address of the first element of the array, numary+0.
- To calculate the address of the last element of the array, the number of elements in the array is first decremented by 1 and then multiplied by 4.
- 5 minus 1 is 4, times 4 is 16, which when added to 104 is 120 in decimal or 114 in hexadecimal.

Indexing Using esi and edi Register (Cont.)

- Here is the MASM code segment to reverse the contents of the given array using the esi and edi registers.

```
mov ecx,n           ; load ecx with contents of n
sar ecx,1           ; divide ecx by 2, number of times to loop
lea esi,numary+0    ; load address of numary into esi
mov edi,esi         ; move contents of esi to edi
mov eax,n           ; load eax with contents of n
dec eax             ; decrement eax by one
sal eax,2           ; multiply eax by 4
add edi,eax         ; add eax to edi for ending address of array
.repeat
mov eax,[esi]       ; move contents where esi is pointing to eax
xchg eax,[edi]      ; exchange eax and where edi is pointing
mov [esi],eax       ; move eax to where edi is pointing
add esi,4           ; add four to esi for next element
sub edi,4           ; subtract four from edi for next element
.untilcxz
```

Indexing Using esi and edi Register (Cont.)

LENGTHOF Operator

- The length of the array can be declared by using the LENGTHOF operator.
- The LENGTHOF operator indicates how many elements are there in an array.
- This operator instructs the assembler to calculate the length of the array at assembly time.
- Then, every time the length of the array is changed, the assembler would recalculate the length of the array.

Indexing Using esi and edi Register (Cont.)

SIZEOF Operator

- The size of the array can be declared by using the SIZEOF operator.
- The SIZEOF operator indicates how many bytes are there in an array.
- This operator instructs the assembler to calculate the size of the array at assembly time.
- Then, every time the size of the array is changed, the assembler would recalculate the size of the array.

Indexing Using esi and edi Register (Cont.)

- Using both the LENGTHOF and SIZEOF operators, the previous code segment could be rewritten as follow.

```
mov ecx,lengthof numary  ; load ecx with length of numary
sar ecx,1                ; divide ecx by 2, of times to loop
lea esi,numary+0         ; load address of numary into esi
mov edi,esi              ; move contents of esi to edi
mov eax,sizeof numary    ; load eax with size of numary
sub eax,4                ; decrement eax by four
add edi,eax              ; add eax to edi for ending address of array
.repeat
mov eax,[esi]            ; move contents where esi is pointing to eax
xchg eax,[edi]           ; exchange eax and where edi is pointing
mov [esi],eax            ; move eax to where edi is pointing
add esi,4                ; add four to esi for next element
sub edi,4                ; subtract four from edi for next element
.untilcxz
```

Sequential Search

- Assuming that the data has already been entered into an array, the number of elements in the array is known, and there are no duplicates in the array. Determine whether the data being searched for is in the array by using sequential search.

Sequential Search (Cont.)

- The equivalent C code would be as follow.

```
int arry[20],n=20,i,number,found;
printf("\n%s","Enter the integer to be found: ");
scanf("%d",&number);
i=0;
found=0;
while(i<n && !found)
   if(number==arry[i])
      found=-1;
   else
      i++;
if (found)
   printf("\n%s\n\n", "The integer was found");
else
   printf("\n%s\n\n","The integer was not found");
```

Sequential Search (Cont.)

- The equivalent MASM code segment would be as follow.

```
            .data
msg1fmt     byte "%s",0
msg2fmt     byte 0Ah,"%s",0Ah,0Ah,0
in1fmt      byte "%d",0
msg1        byte "Enter the integer to be found: ",0
msg2        byte "The integer was found",0
msg3        byte "The integer was not found",0
arry        sdword 20 dup(?)
n           sdword 20
number      sdword ?
found       sbyte ?

            .code
            INVOKE printf, ADDR msg1fmt,ADDR msg1
            INVOKE scanf, ADDR in1fmt, ADDR number
            mov ebx,0             ; initialize ebx to 0
            mov ecx,0             ; initialize ecx to 0
            mov edx,number        ; load edx with number
            mov found,0           ; initialize found to 0
            .while(ecx<n && !found)
            .if(edx==arry[ebx])
            mov found,-1          ; set found to -1
            .else
            add ebx, 4             ; increment ebx by 4
            .endif
            inc ecx                ; increment ecx by 1
            .endw
            .if(found)
            INVOKE printf, ADDR msg2fmt, ADDR msg2
            .else
            INVOKE printf, ADDR msg2fmt, ADDR msg3
            .endif
```

Complete Program: Implementing a Queue

- The implementation of a queue provides an excellent opportunity to demonstrate the use of an array and indexing.
- Consider the C and MASM programs for the given situations. The main program uses a sentinel-controlled loop to continue to iterate until the letter s is input, which stands for stop. Then for each iteration of the loop, it checks for the letter e for enqueue or d for dequeue, otherwise an appropriate error message is output.

Complete Program: Implementing a Queue

- The equivalent C code would be as follow.

```
#include <stdio.h>
const int n=3;
int queue[3],number,front=0,rear=0,count=0;
char command;
int main() {
   void enqueue();
   void dequeue();
   printf("\n%s","Enter a command, e, d, or s: ");
   scanf("%s",&command);
   while (command != 's'){
      if (command=='e'){
         printf("\n%s","Enter a positve integer: ");
         scanf("%d",&number);
         enqueue();
      }
      else
         if (command=='d'){
            dequeue();
            if (number>0)
               printf("\n%s%d\n","The integer is: ",number);
            }
            else
               printf("\n%s","Invalid entry, try again");
            printf("\n%s","Enter a command, e, d, or s: ");
            scanf("%s",&command);
         }
         printf("\n");
         return 0;
}
```

```
void enqueue(){
   if (count<n){
      count++;
      queue[rear]=number;
      rear=(rear+1)%n;
   }
   else
      printf("\n%s\n","Error: Queue is full");
}
void dequeue(){
   if (count>0){
      count--;
      number=queue[front];
      front=(front+1)%n;
   }
   else{
      printf("\n%s\n","Error: Queue is empty");
      number =-1;
   }
}
```

Complete Program: Implementing a Queue

- The equivalent MASM code would be as follows.

```
            .listall
            .386
            .model flat,c
            .stack 100h
scanf       PROTO arg2:Ptr Byte, inputlist:VARARG
printf      PROTO arg1:Ptr Byte, printlist:VARARG

            .data
in1fmt      byte "%s",0
in2fmt      byte "%d",0
msg1fmt     byte 0Ah,"%s",0
msg3fmt     byte 0Ah,"%s%d",0Ah,0
msg4fmt     byte 0Ah,0
errfmt      byte 0Ah,"%s",0Ah,0
msg1        byte "Enter a command, e, d, or s: ",0
msg2        byte "Enter a positive integer: ",0
msg3        byte "The integer is: ",0
errmsg1     byte "Error: Invalid entry, try again",0
errmsg2     byte "Error: Queue is full",0
errmsg3     byte "Error: Queue is empty",0
queue       sdword 3 dup(?)
command     sdword ?
number      sdword ?
count       sdword 0

            .code
enqueue     macro
            .if count < lengthof queue
             inc count              ; increment count
             mov eax,number         ; load eax with number
             mov [edi],eax          ; store eax in rear
             mov eax,edi            ; copy edi (rear) to eax
             sub eax,offset queue   ; subtract address of queue
             add eax,4              ; increment eax by 4
             cdq                    ; convert double to quad
             mov ecx,sizeof queue   ; get size of queue (bytes)
             idiv ecx               ; divide
             mov edi,offset queue   ; load address in rear
             add edi,edx            ; add remainder to rear
             .else
             INVOKE printf, ADDR errfmt, ADDR errmsg2
             .endif
             endm

dequeue     macro
            .if count > 0
            dec count               ; decrement count
            mov eax,[esi]           ; load eax from front
            mov number,eax          ; store eax in number
            mov eax,esi             ; copy esi (front) to eax
            sub eax,offset queue    ; subtract address of queue
            add eax,4               ; increment eax by 4
            cdq                     ; convert double to quad
            mov ecx, sizeof queue   ; get size of queue (bytes)
            idiv ecx                ; divide
            mov esi,offset queue    ; load address in front
            add esi,edx             ; add remainder to front
            .else
            INVOKE printf, ADDR errfmt, ADDR errmsg3
            mov number,-1           ; store -1 (flag) in number
            .endif
            endm

main        proc
            mov edi,offset queue+0 ; use edi as front of queue
            mov esi,offset queue+0 ; use esi as rear of queue
            INVOKE printf, ADDR msg1fmt, ADDR msg1   ; priming
            INVOKE scanf, ADDR in1fmt, ADDR command  ; read
            .while command != "s"  ; while not stop
            .if command=="e"       ; enqueue?
            INVOKE printf, ADDR msg1fmt, ADDR msg2
            INVOKE scanf, ADDR in2fmt, ADDR number
            enqueue                ; enqueue number
            .elseif command=="d"   ; dequeue?
            Dequeue                ; deque number
            .if number >0          ; not -1 (flag)?
            INVOKE printf, ADDR msg3fmt, ADDR msg3, number
            .endif
            .else
            INVOKE printf, ADDR errfmt, ADDR errmsg1
            .endif
            INVOKE printf, ADDR msg1fmt, ADDR msg1
            INVOKE scanf, ADDR in1fmt, ADDR command
            .endw
            INVOKE printf, ADDR msg4fmt
            ret
main        endp
            end
```

Complete Program: Implementing Selection Sort

- Implementing sorting provides an excellent opportunity to examine nested loops, ifs and the use of esi and edi registers.

```
            .listall
            .386
            .model flat,c
            .stack 100h
scanf       PROTO arg2:Ptr Byte, inputlist:VARARG
printf      PROTO arg1:Ptr Byte, printlist:VARARG

            .data
msg1fmt     byte 0Ah,"%s",0
msg2fmt     byte "%s",0
msg3fmt     byte 0Ah,"%s",0Ah,0Ah,0
msg4fmt     byte "   %d",0Ah,0
msg5fmt     byte 0Ah,0
in1fmt      byte "%d",0

msg 1       byte "Enter the number of integers to be input: ",0
msg2        byte "Enter an integer: ",0
 msg3       byte "Sorted",0
 n          sdword ?
 arry       sdword 20 dup(?)
 temp       sdword ?

            .code
main        proc
            INVOKE printf,ADDR msg1fmt,ADDR msg1
            INVOKE scanf,ADDR in1fmt,ADDR n
            INVOKE printf,ADDR msg5fmt
            .if n>0                 ; if n <= 0, don't continue
            mov ecx,n               ; load ecx with n
            mov edi,offset arry+0   ; load address of arry into edi
            .repeat
            push ecx                ; save ecx
            INVOKE printf,ADDR msg2fmt,ADDR msg2
            INVOKE scanf,ADDR in1fmt,ADDR [edi]
            add edi,4               ; incrment edi by 4
            pop ecx                 ; restore ecx
            .untilcxz
            .if n>1                 ; check >1 elements in array
            mov ecx,n               ; load ecx with n
            dec ecx                 ; loop n-1 times
            mov esi,offset arry+0   ; load esi with address of arry
            .repeat
            push ecx                ; save ecx
            push esi                ; save address, esi now smallest
            mov edi,esi             ; load address of esi in edi
            add edi,4               ; move edi to the next element
            .repeat
            mov eax,[esi]           ; move smallest to eax to compare
            .if [edi]<eax           ; compare smallest to next
            mov esi,edi             ; save the new smallest in esi
            .endif
            add edi,4               ; move to next element to compare
            .untilcxz
            mov edi,esi             ; edi points to smallest element
            pop esi                 ; esi points to the start element
            mov eax,[esi]           ; move start element to temp
            xchg eax,[edi]          ; exchange start and smallest
            mov [esi],eax           ; move smallest back to start
            add esi,4               ; move start index to next
            pop ecx                 ; restore ecx to be decremented
            .untilcxz
            .endif
            INVOKE printf, ADDR msg3fmt, ADDR msg3
            mov ecx, n              ; load ecx with n
            mov esi,offset arry+0   ; load esi with address of arry
            .repeat
            push ecx                ; save ecx
            mov eax,[esi]           ; load eax with element from arry
            mov temp,eax            ; store eax in temp for output
            INVOKE printf, ADDR msg4fmt, temp
            add esi.4               : increment esi to next element
            pop ecx                 ; restore ecx
            .untilcxz
            INVOKE printf, ADDR msg5fmt
            .endif
            ret
main        endp
            end
```

Summary

- The DUP operator allows for the declaration of large initialized or uninitialized arrays.
- The ebx register can be used as an index for an array.
- When dealing with arrays of sdword, remember to increase by 4 instead of 1 because signed double word takes up 4 bytes.
- Use square brackets [] around the ebx, esi, and edi registers, not to get the contents of the register but rather to get the contents of the memory location to which they are indexing or pointing.

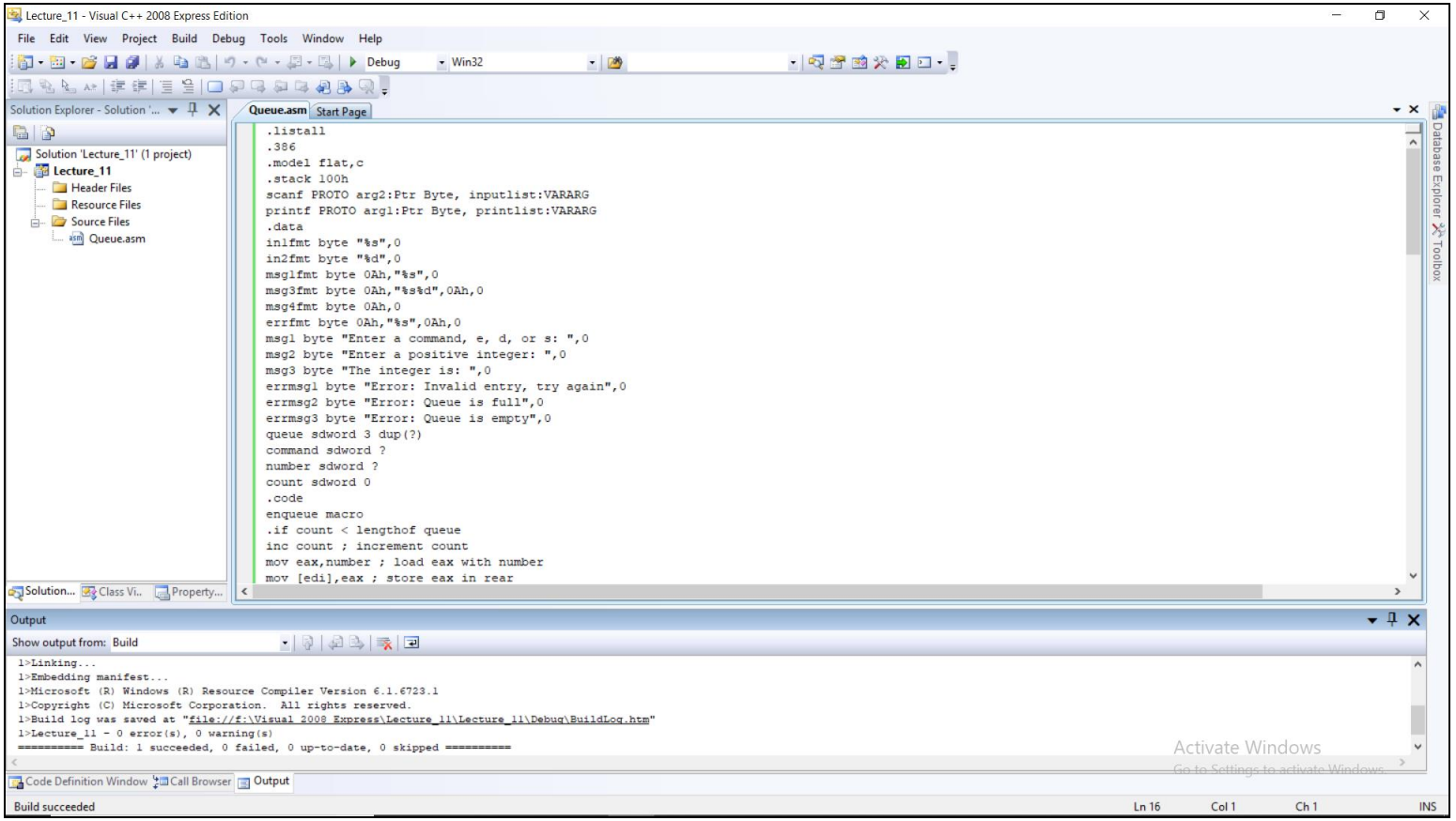
Microprocessor Programming

Practical Works

Implementing Queue and Selection Sort

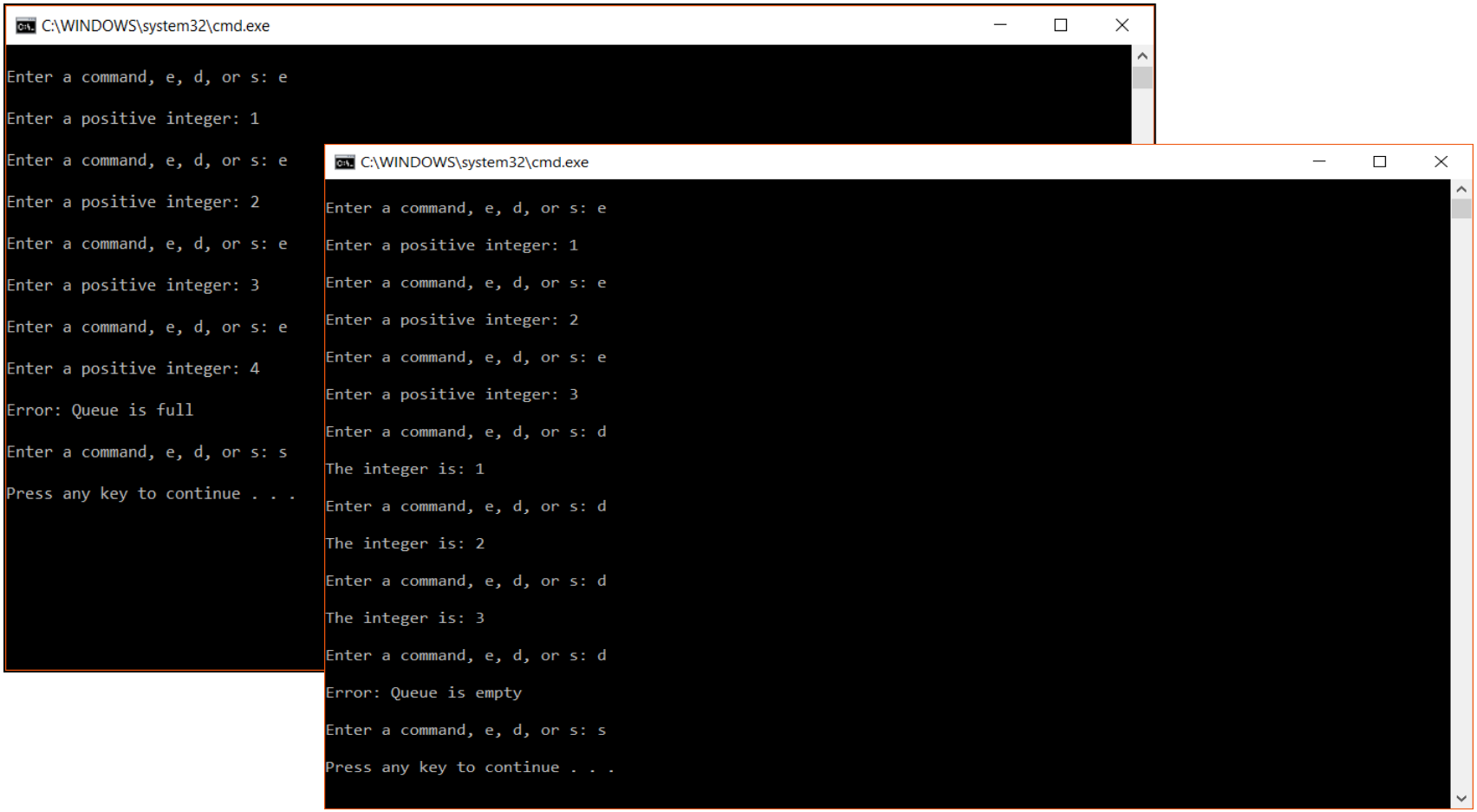
Implementing a Queue

- The implementation of the given MASM program for a queue is described below:



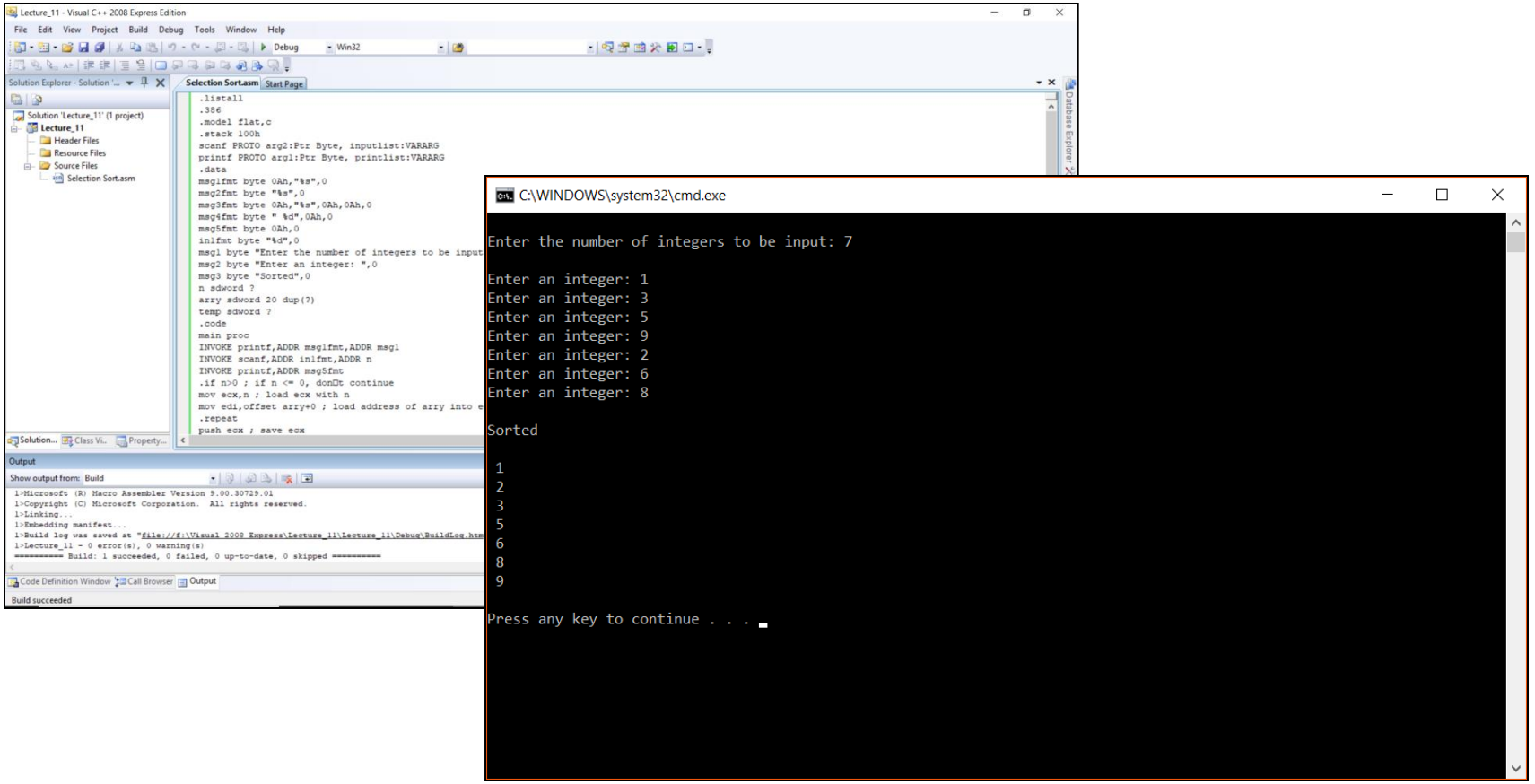
Implementing a Queue

- The outputs of the given program are described below:



Implementing a Selection Sort

- The implementation of the given MASM program for selection sort and its output are described below:



Microprocessor Programming

Practical Assignments (Instructions)

Assignment 1

- Write a complete assembly program using the code segment given below and show the output message.

```
                .data
msg1fmt         byte 0Ah,"%s",0
msg2fmt         byte 0Ah,"%s",0Ah,0Ah,0
msg3fmt         byte " %d",0Ah,0Ah, 0
in1fmt          byte "%d",0
msg1            byte "Enter the number of integers to be input: ",0
msg2            byte "Enter an integer: ",0
msg3            byte "Reversed",0
msg4            byte "No data entered."
n               sdword ?
arry            sdword 20 dup(?)
                .code
                INVOKE printf, ADDR msg1fmt,ADDR msg1
                INVOKE scanf, ADDR in1fmt, ADDR n
                mov ecx,n                       ; initialize ecx to n
                mov ebx,0                       ; initialize ebx to 0
                .if ecx>0
                .repeat
                push ecx                        ; save ecx
                INVOKE printf, ADDR msg1fmt, ADDR msg2
                INVOKE scanf, ADDR in1fmt, ADDR arry[ebx]
                pop ecx                         ; restore ecx
                add ebx,4                       ; increment ebx by 4
                .untilcxz
                INVOKE printf, ADDR msg2fmt, ADDR msg3
                mov ecx,n                       ; initialize ecx to n
                sub ebx,4                       ; subtract 4 from ebx
                .repeat
                push ecx                        ; save ecx
                INVOKE printf, ADDR msg3fmt,arry[ebx]
                pop ecx                         ; restore ecx
                sub ebx,4                       ; decrement ebx by 4
                .untilcxz
                .else
                INVOKE printf, ADDR msg2fmt, ADDR msg4
                .endif
```

Assignment 2

- Write a complete assembly program using the code segment given below and show the output messages.

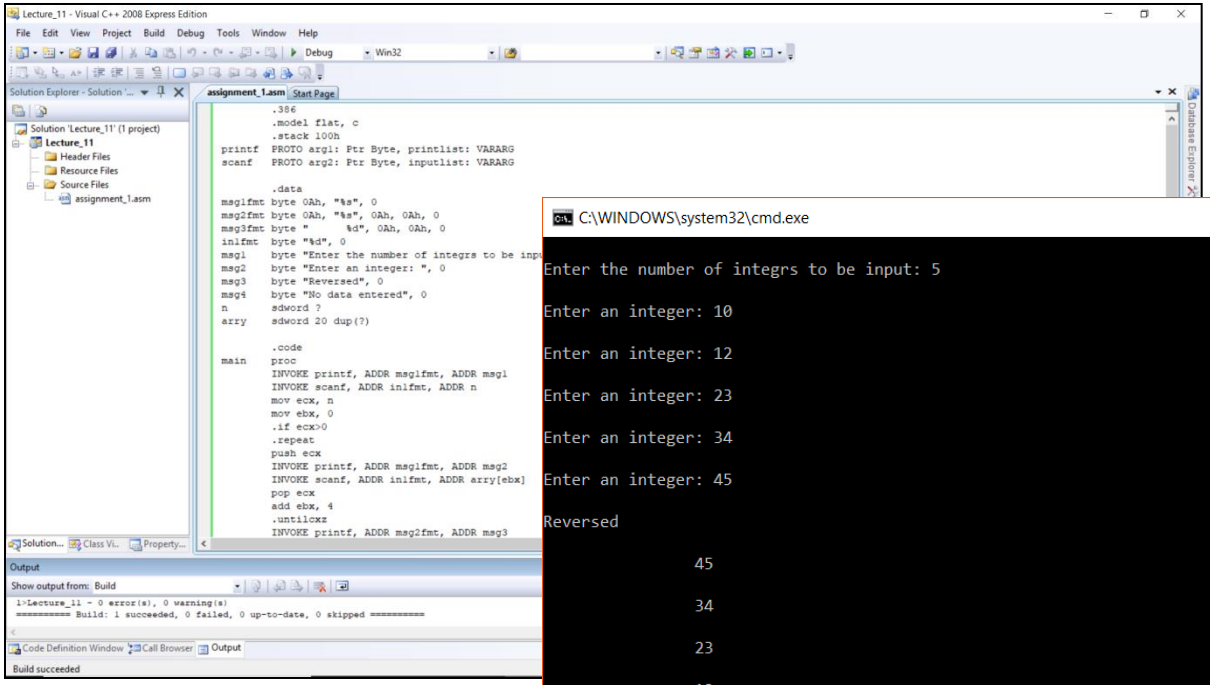
```
.data
msg1fmt byte "%s",0
msg2fmt byte 0Ah,"%s",0Ah,0Ah,0
in1fmt byte "%d",0
msg1 byte "Enter the integer to be found: ",0
msg2 byte "The integer was found",0
msg3 byte "The integer was not found",0
arry sdword 1,2,3,4,5
n sdword 5
number sdword ?
found sbyte ?
.code
INVOKE printf, ADDR msg1fmt,ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR number
mov ebx,0           ; initialize ebx to 0
mov ecx,0           ; initialize ecx to 0
mov edx,number      ; load edx with number
mov found,0         ; initialize found to 0
.while(ecx<n && !found)
.if(edx==arry[ebx])
mov found,-1        ; set found to -1
.else
add ebx, 4          ; increment ebx by 4
.endif
inc ecx             ; increment ecx by 1
.endw
.if(found)
INVOKE printf, ADDR msg2fmt, ADDR msg2
.else
INVOKE printf, ADDR msg2fmt, ADDR msg3
.endif
```

Microprocessor Programming

Practical Assignments (Report)

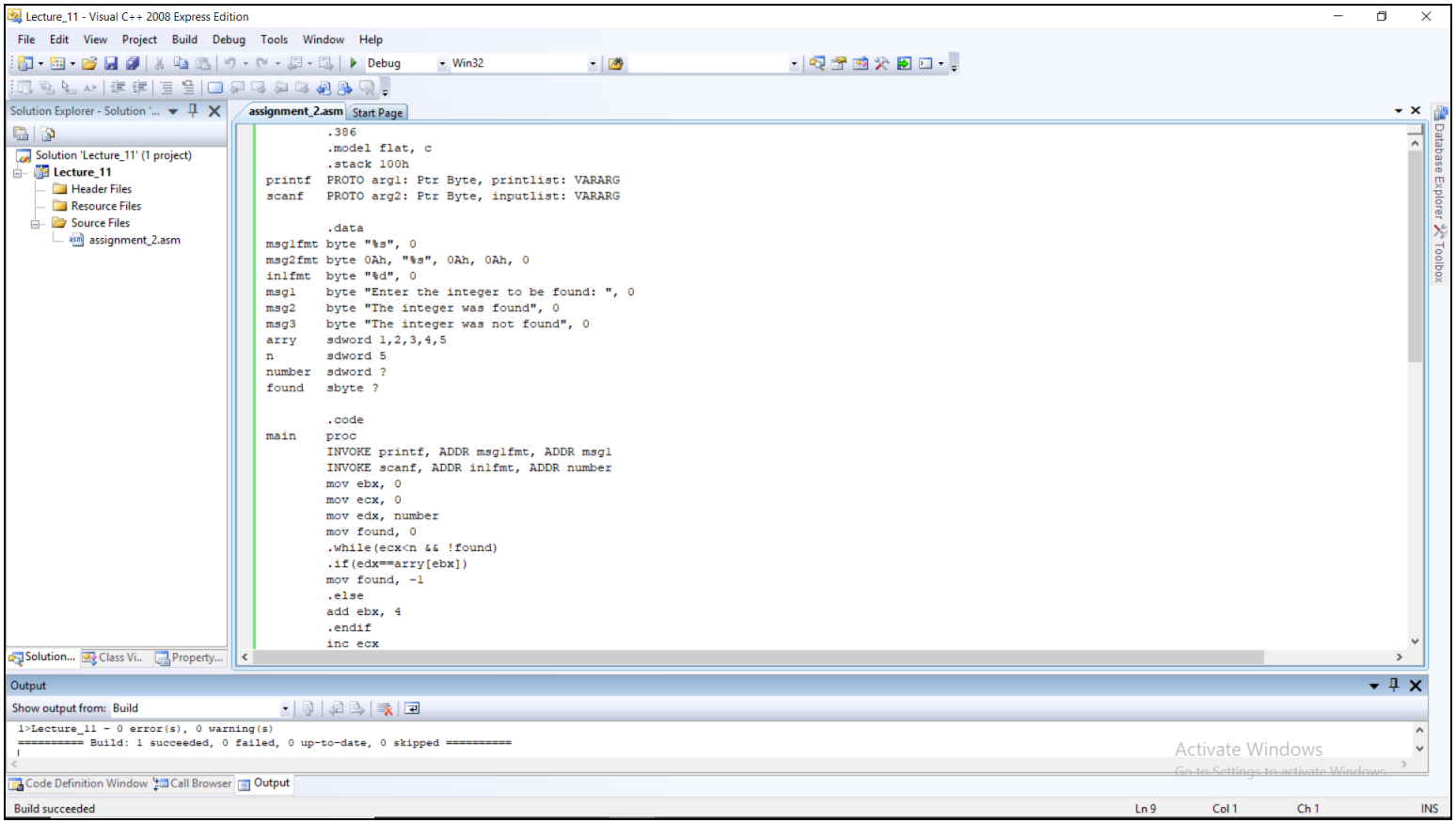
Assignment 1

- A complete assembly program and its output are described below:



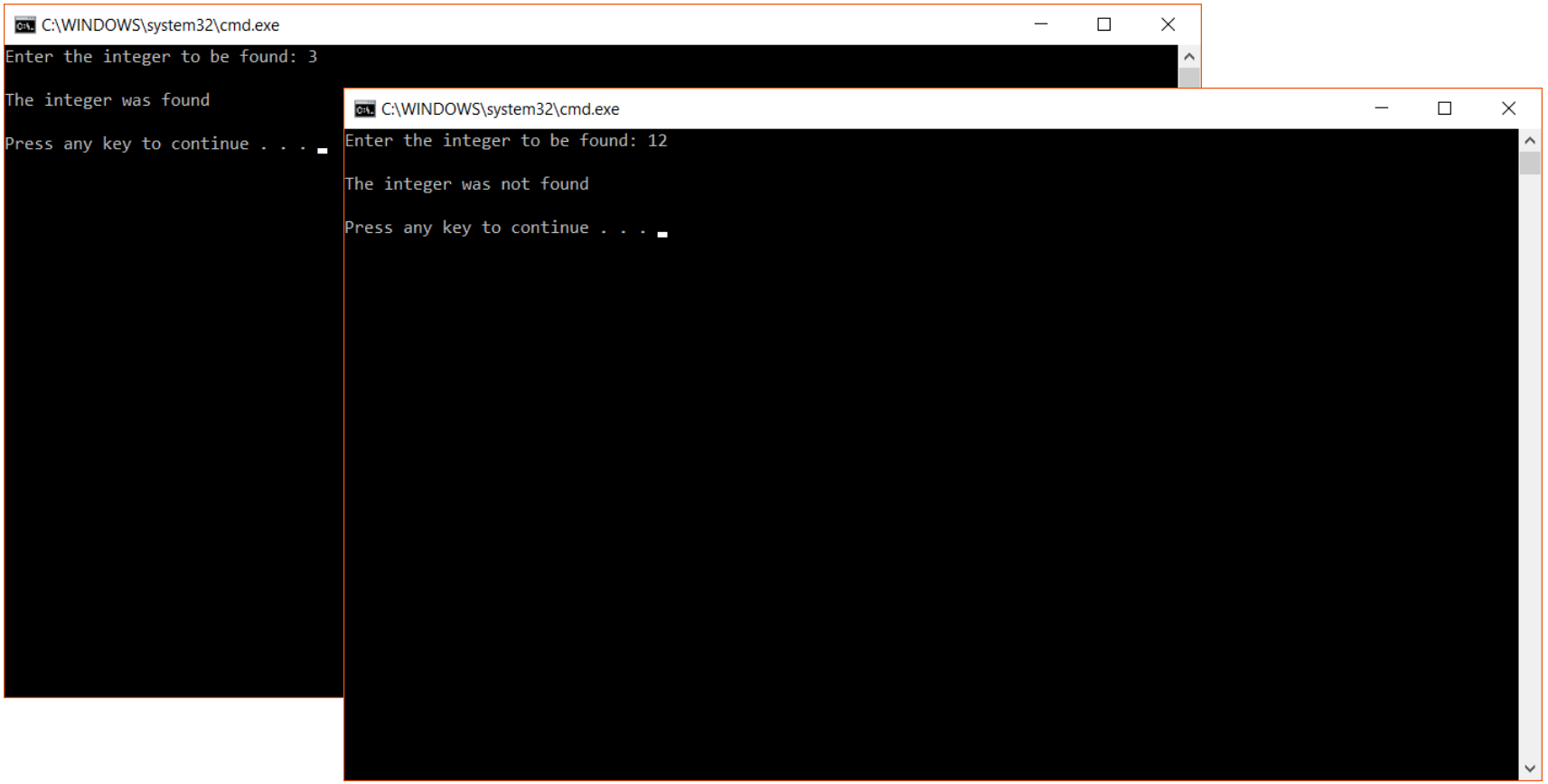
Assignment 2

- A complete assembly program is described below:



Assignment 2

- The outputs for the complete assembly program are described below:



Next Lecture

- Strings
- Array of Strings

Thank You