CMPE 310

Lab 5 - Processing Integers

Outline

- NASM Review
  - What is a Stack Frame/ Call Frame?
  - Macros

- Homework 3
  - Homework 3 Description
  - Input File Format
  - C-Functions
  - GetCommandLine
  - mine.inc

What is a Stack Frame/ Call Frame?

Stack Frame is used to protect data pushed into the stack

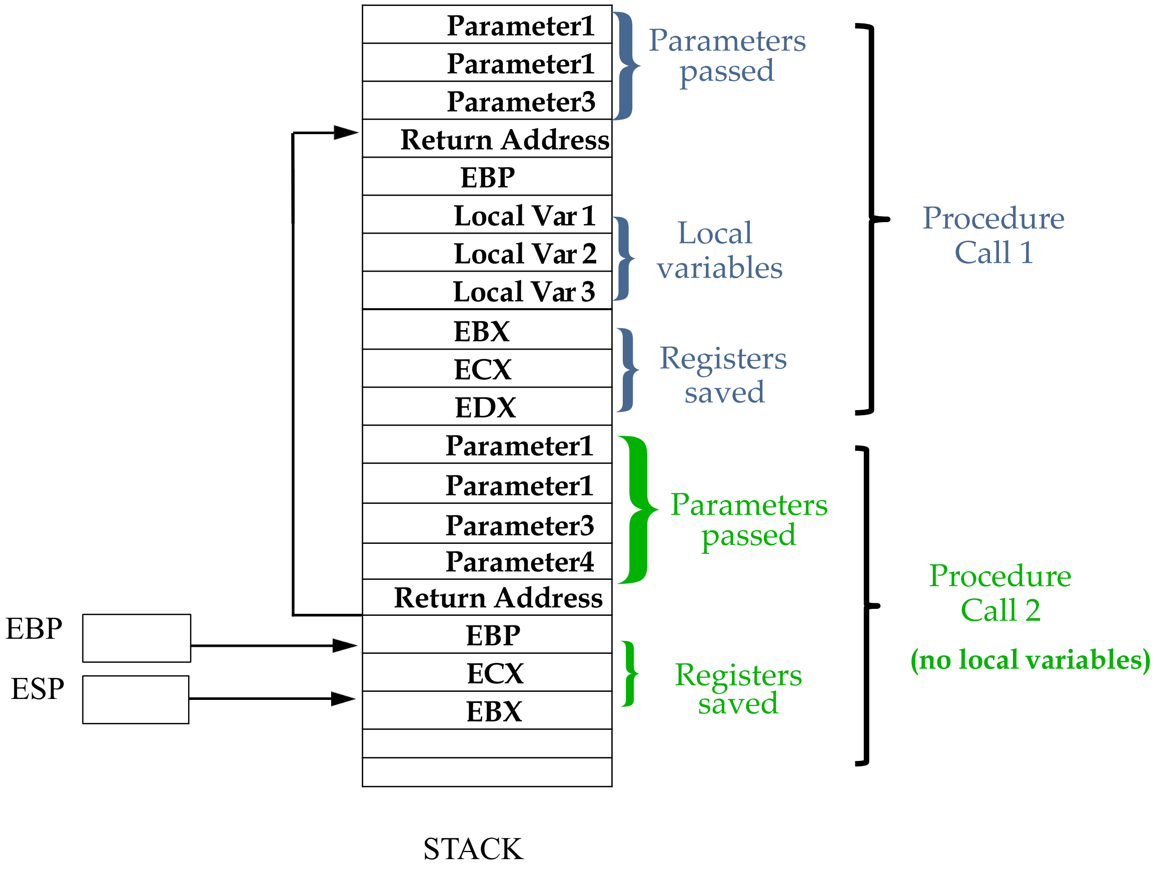
    move EBP to ESP.

    Prevents POPing.

Can reach data stored in a stack frame by dereferencing EBP with an offset.

Call Frames

One call frame created per procedure call



Setting up Call Frames

```
GetCommandLine:
        Enter 0                        (1)
        Push_Regs ebx, ecx, edx        (2)
```

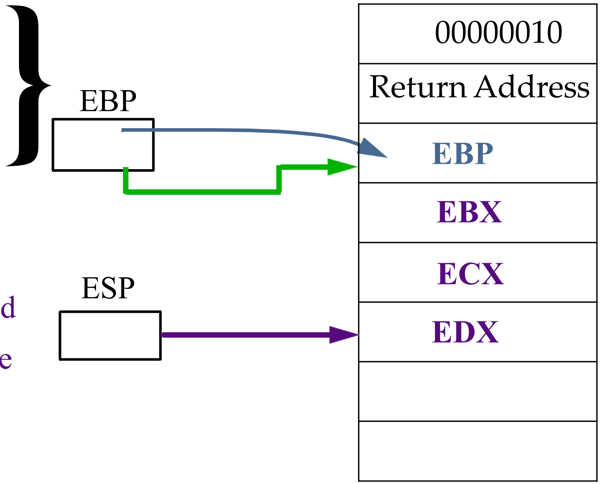
```
%macro Enter 1
    push ebp
    mov ebp, esp
    sub esp, %1
%endmacro
```

(1) Push EBP

Move ESP into EBP
i.e. EBP points to the pushed EBP

Allocate space for local variables
(none in this example)

(2) Push the registers that are to be saved
EBX, ECX and EDX in this example



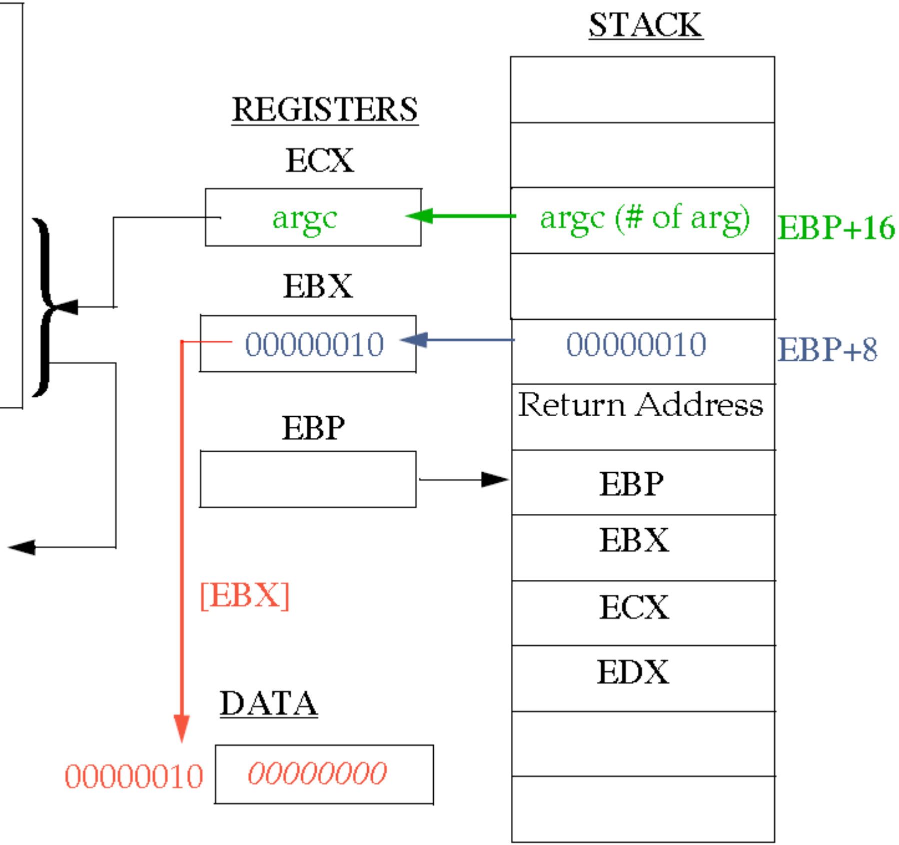
Reading Arguments

```
mov ebx, [ebp + 8]
mov [ebx], dword 0

mov ecx, [ebp + 16]
cmp ecx, 2
    if ne
        jmp gcl_done
    endif
```

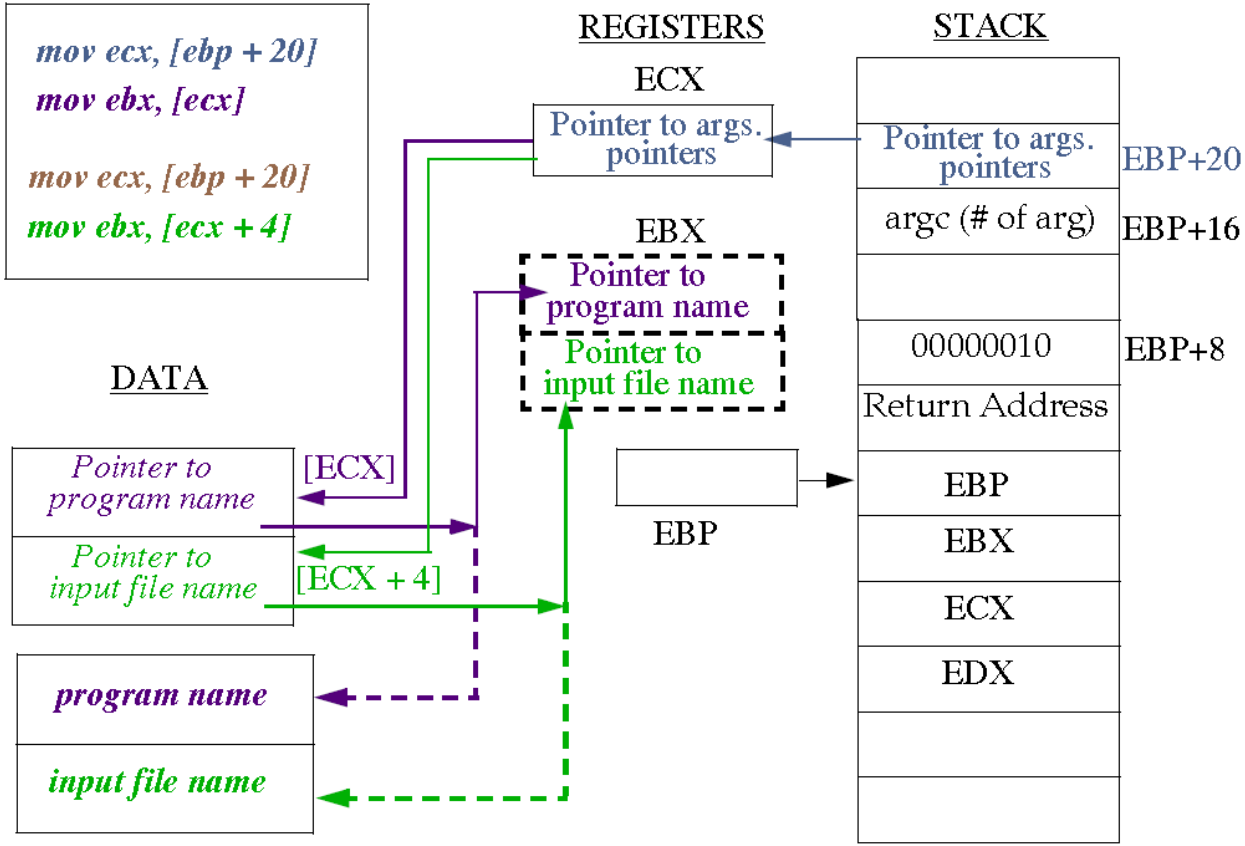
Exactly 2 arguments required

Program name and input file name

ELSE ERROR!!!



Reading Arguments



```
mov ecx, [ebp + 20]
mov ebx, [ecx]

mov ecx, [ebp + 20]
mov ebx, [ecx + 4]
```

REGISTERS

ECX

Pointer to args. pointers

EBX

Pointer to program name

Pointer to input file name

EBP

STACK

Pointer to args. pointers EBP+20

argc (# of arg) EBP+16

00000010 EBP+8

Return Address

EBP

EBX

ECX

EDX

DATA

Pointer to program name [ECX]

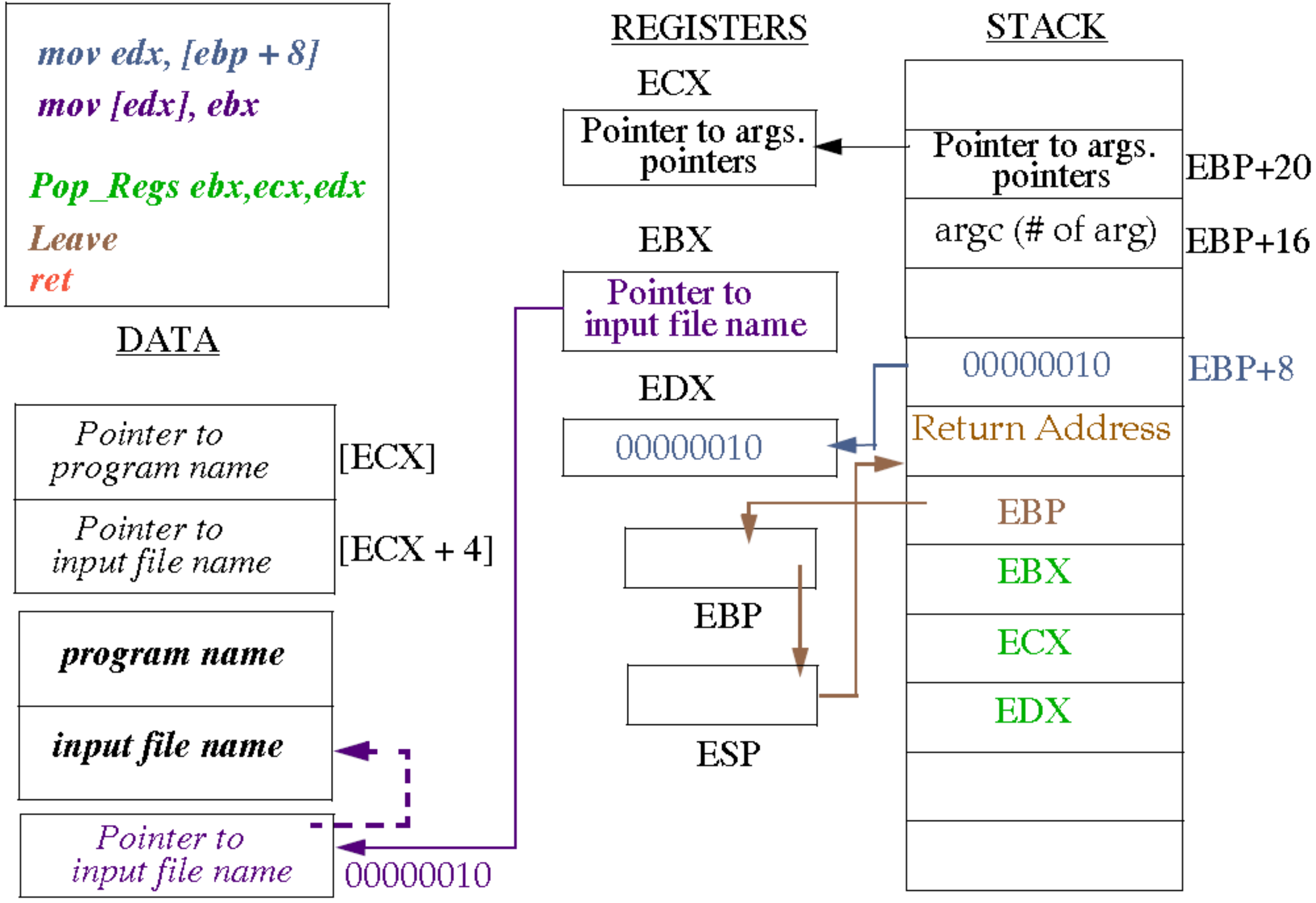
Pointer to input file name [ECX + 4]

program name

input file name

Get argument and Return

```
mov edx, [ebp + 8]
mov [edx], ebx

Pop_Regs ebx,ecx,edx
Leave
ret
```



DATA

Pointer to program name [ECX]

Pointer to input file name [ECX + 4]

program name

input file name

Pointer to input file name 00000010

REGISTERS

ECX: Pointer to args. pointers

EBX: Pointer to input file name

EDX: 00000010

EBP

ESP

STACK

Pointer to args. pointers EBP+20

argc (# of arg) EBP+16

00000010 EBP+8

Return Address

EBP

EBX

ECX

EDX

Procedure Calls (Steps Recap)

Caller: Before Call

- Save registers that are needed (for C functions save EAX, ECX, EDX)
- Push arguments, last first
- CALL the function

Callee:

- Save caller's EBP and set up callee stack frame (ENTER macro)
- Allocate space for local variables and temporary storage
- Save registers as needed (C functions save EBX, ESI, EDI)
- Perform the task
- Store return value in EAX
- Restore registers (C functions restore EBX, ESI, EDI)
- Restore caller's stack frame (LEAVE macro)
- Return

Caller: After Return

- POP arguments, get return value in EAX, restore registers (for C EAX, ECX, EDX)

Macros

Single-line Macros:

```
%define ctrl 0x1F &                         ;Definitions
%define param(a,b) ((a)+(a)*(b))
```

Can be used as:

```
        mov byte [param(2,ebx)], ctrl ‘D’
```

Which expands to:

```
        mov byte [(2)+(2)*(ebx)], 0x1F & ‘D’
```

Note that expansion occurs at invocation time, not at definition time, e.g.

```
%define a(x) 1+b(x)                  ;b(x) used before it is
%define b(x) 2*x                     ;defined here.
```

Used as:

```
        mov ax, a(8)
```

Expands to:

```
        mov ax, 1+2*8
```

Macros

Overloading macros is allowed.

```
%define foo(x) 1+x                  ;Single arg definition
%define foo(x,y) 1+x*y              ;Double arg definition
```

Undefining macros:

```
%undef foo
```

Multi-line Macros:

```
%macro prologue 1
        push ebp
        mov ebp, esp
        sub esp, %1
%endmacro
```

And use as:

```
myfunc: prologue 12
```

Expands to:

```
myfunc: push ebp
        mov ebp, esp
        sub esp, 12
```

Macros

Conditional assembly:

Given the macro (21h is a DOS interrupt):

```
%macro writefile 2+              ;Greedy macro params
          jmp %%endstr           ;%% defines macro-local
%%str:    db %2                  ;labels which are different
%%endstr: mov dx, %%str          ;each time the macro is
          mov cx, %%endstr-%%str ;invoked.
          mov bx, %1
          mov ah, 0x40
          int 0x21
%%endmacro
```

And the call:

```
%ifdef DEBUG
          writefile 2, “I'm here”, 13, 10
%endif
```

Using the command-line option -dDEBUG, expands the macro otherwise it is left out (similar to C).

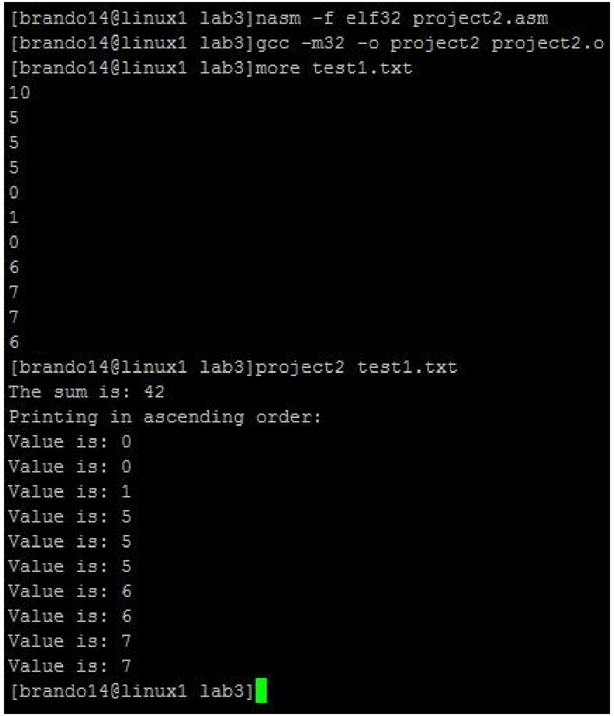
Note that “I'm here”, 13, 10 is substituted in for %2 in the above code.

Homework 3 Description

- Read in a set of integers from an input file to a memory array.
    - The first line of the input file will contain the number of integers in the file.
    - There will be a maximum of 1000 integers in the input file (so a maximum of 1001 lines).
    - The input file name is to be read from the command line (use the GetCommandLine function that we are providing).
    - Use the C-functions fopen and fscanf to open and read from the input file.
- Compute the sum of the integers that you have read in and print the sum to the terminal.
    - Just like in the last lab, we will use printf for printing integers to the terminal.
- Sort the array and print the sorted contents in descending order.

Input File Format

Example of an input file that you would use to test your homework 3 as well as the corresponding output (shown here in ascending order):



```
[brando14@linux1 lab3]nasm -f elf32 project2.asm
[brando14@linux1 lab3]gcc -m32 -o project2 project2.o
[brando14@linux1 lab3]more test1.txt
10
5
5
5
0
1
0
6
7
7
6
[brando14@linux1 lab3]project2 test1.txt
The sum is: 42
Printing in ascending order:
Value is: 0
Value is: 0
Value is: 1
Value is: 5
Value is: 5
Value is: 5
Value is: 6
Value is: 6
Value is: 7
Value is: 7
[brando14@linux1 lab3]
```

It wouldn't be a bad idea to use a higher level language that you are more comfortable with to generate large input test files.

C-Functions

printf

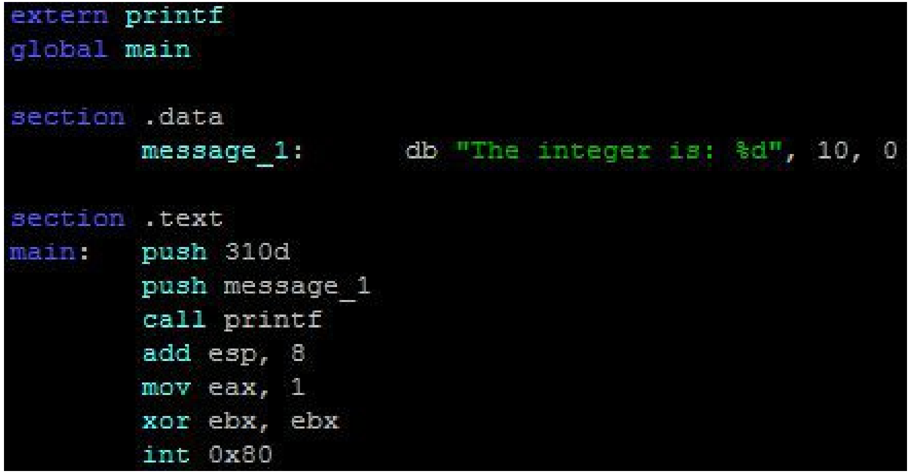
- C library function that sends formatted output to stdout.
- To use printf, you must have the line “extern printf” somewhere in your program.
- The “%d” in your output string tells printf where to place the data that you have passed it and how to format it (decimal in this case).

- The value 10 at the end of the message represents the new line character.

- The value 0 represents the null character.

- The assembly code on the right in C would look something like:

  printf(“The integer is: %d”, 310)

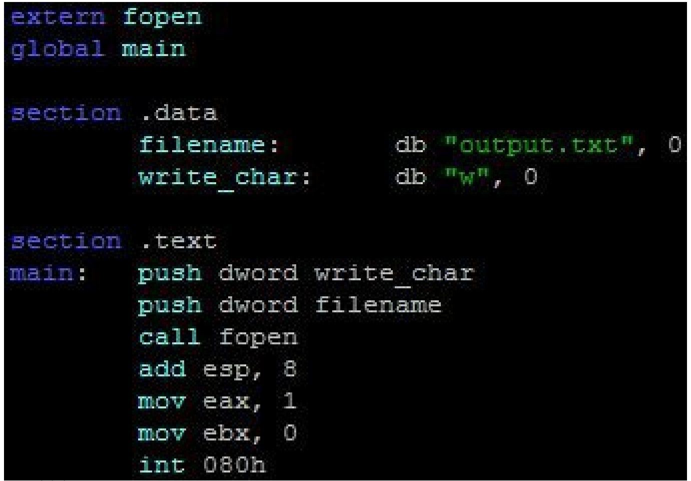


```
extern printf
global main

section .data
        message_1:      db "The integer is: %d", 10, 0

section .text
main:   push 310d
        push message_1
        call printf
        add esp, 8
        mov eax, 1
        xor ebx, ebx
        int 0x80
```

C-Functions

fopen

C library function that opens or creates the file given by filename using the given mode.
Must include “extern fopen” in your program.



```
extern fopen
global main

section .data
        filename:       db "output.txt", 0
        write_char:     db "w", 0

section .text
main:   push dword write_char
        push dword filename
        call fopen
        add esp, 8
        mov eax, 1
        mov ebx, 0
        int 080h
```

The “mode” here is the file access mode.
The “filename” in the example to the right is the address where our “output.txt” string is stored.

For this assignment we are getting the filename from the command line (using GetCommandLine). GetCommandLine will place the address of the input file into the filename variable, so

push dword filename

The assembly code on the right in C would look something like:

fopen(“output.txt”, “w”)

C-Functions

fopen continued

After calling fopen, a file pointer is returned in eax.

This file pointer is what you will be using in fscanf.

If there was an error with the fopen call, NULL will be returned in eax.

C-Functions

fscanf

C library function that reads formatted input from a stream (our input file in this case).

The “stream” is the file pointer that we get from fopen.

The “format” here tells fscanf how to interpret each line (decimal, character, hex, etc.).

The line that is read in will be stored in the first variable pushed to the stack.

Each consecutive call to fscanf automatically proceeds to the next line of the input file.

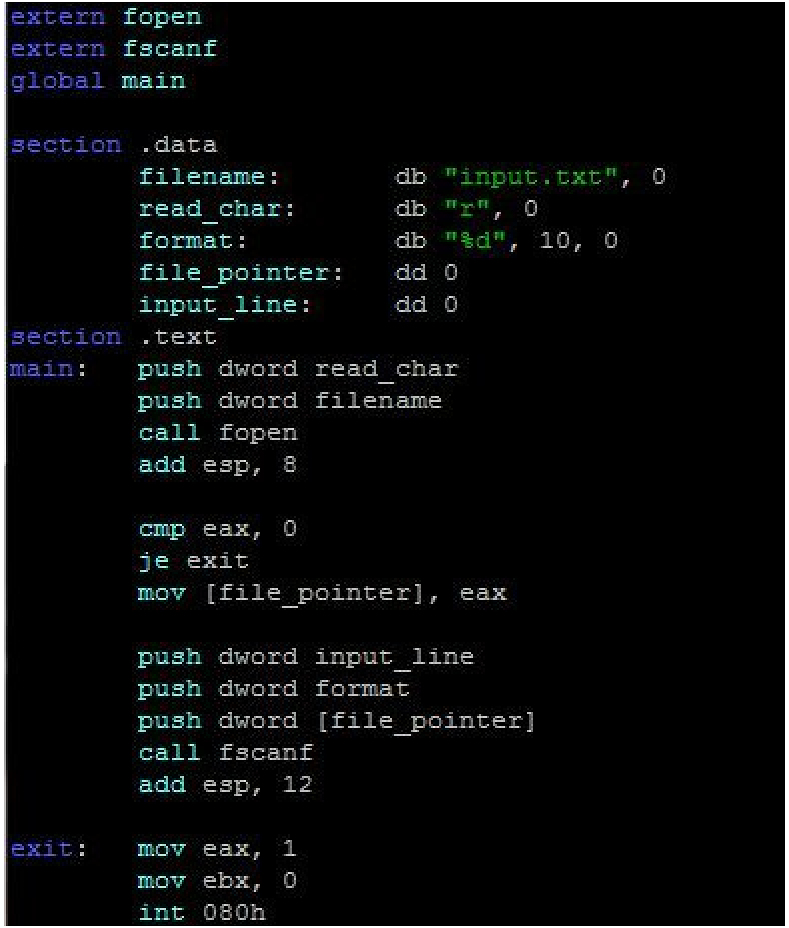
C-Functions

fscanf

The example on the right opens a file in read only mode, performs an error check on the call to fopen, and reads in the first line of the input file and stores it in “input_line”.

The code in C would look something like:

file_ptr = fopen(“input.txt”, “r”)
fscanf(file_ptr, “%d”, input_line)



```
extern fopen
extern fscanf
global main

section .data
        filename:       db "input.txt", 0
        read_char:      db "r", 0
        format:         db "%d", 10, 0
        file_pointer:   dd 0
        input_line:     dd 0
section .text
main:   push dword read_char
        push dword filename
        call fopen
        add esp, 8

        cmp eax, 0
        je exit
        mov [file_pointer], eax

        push dword input_line
        push dword format
        push dword [file_pointer]
        call fscanf
        add esp, 12

exit:   mov eax, 1
        mov ebx, 0
        int 080h
```