Laboratory 10
Data Structures Representation
Computer Science 240

Writing X86 Code Directly

Assembly Directives
Begin with a dot and indicate structural information useful to the assembler, linker, or debugger.

- indicates label main is a global symbol that can be accessed by other code modules.

```
.globl main
```

- store instructions following directive in the text segment of memory

```
.text
```

- store declared data following directive in the data segment of memory

```
data
```

- allocate space for an 8 byte value and initialize to 0

```
i: .quad 0
```

- allocate space and initialize to specified null-terminated string

```
str:.string "Enter a value"
```

Lab Assignment

**simple.c: (C code)**

```c
#include <stdio.h>

long total = 0;

int sum(int x, int y) {
  int t = x + y;
  total += t;
  return t;
}

int main() {
  int x = 2;
  int y = 3;
  printf("Sum = %d\n", sum(x, y));
  printf("Total = %d\n", total);
  return 0;
}
```

**simple.s: (X86 code)**

```asm
.data //use the data segment of memory
.globl  total  //total is a global variable

long total = 0;

.total: .quad  0  //8 bytes with initial value 0
fstr1: .string "Sum = %d\n"
fstr2: .string "Total = %d\n"

.text
.globl main

main:
    mov  $0x3,%esi
    mov  $0x2,%edi
    call sum
    mov  %eax,%esi
    mov  $fstr1,%edi
    mov  $0x0,%eax
    call printf

    mov  $total,%esi
    mov  $fstr2,%edi
    mov  $0x0,%eax
    call printf

    mov  $0x0,%eax
    ret
```

I
#include <stdio.h>

int z;

int square(int n) {
    return n*n;
}

int main() {
    int x = square(3);
    int y = square(4);
    z = x + y;
    printf("Calculation produces %d\n", z);
    return 0;
}
One-dimensional arrays

Different languages use different implementations at the machine level to represent data structures.

In Java, arrays are actually implemented as arrays of addresses (pointers) to the elements, which are stored elsewhere in memory (not necessarily in contiguous locations).

In C, the elements of the array are stored in a contiguous block, starting at the base address of the array.

In the C model,

\[
\text{address of element in array} = \text{base address} + \text{element size} \times \text{index}
\]

If the size of the element is limited to 1, 2, or 4 bytes, what is another more efficient way to accomplish the multiplication?

In C, to define some arrays of 8 elements of different sizes:

```c
long qelements[] = {0xF, 0xE, 0xD, 0xC};
int elements[] = {0x1, 0x3, 0x5, 0x7, 0x9, 0x11, 0x13, 0x15};
short welements[] = {0x23, 0x25, 0x27, 0x29, 0x31, 0x33, 0x35, 0x37}
char belements[] = {0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90}
```

The equivalent in X86 is:

```
.data
qelements: .quad 0xF,0xE,0xD,0xC
elements: .long 0x1, 0x3, 0x5, 0x7, 0x9, 0x11,0x13,0x15
welements: .word 0x23,0x25,0x27,0x29,0x31,0x33,0x35,0x37
belements: .byte 0x20,0x30,0x40,0x50,0x60,0x70,0x80,0x90
```

The contents of memory starting at `qelements` displayed using `gdb` would look something like this:

```
0x0049700 <qelements>:    0x00000000 0x0000000F 0x00000000 0x0000000E
0x0849714 <qelements+16>: 0x00000000 0x0000000D 0x00000000 0x0000000C
0x0049724 <elements >:   0x00000001 0x00000003 0x00000005 0x00000007
0x0049734 <elements+16>:  0x00000009 0x00000011 0x00000013 0x00000015
0x0049744 <welements>:    0x00250023 0x00290027 0x00330031 0x00370035
0x 049754 <belements >:   0x50403020 0x90807060
```
Two-dimensional arrays

In C, when nested array of arrays are used, each row is stored contiguously in memory (row-major format), and the address of an element can be calculated by the following formula (size of row is the number of columns in a row):

\[
\text{address of element}[\text{row}][\text{col}] = \\
\quad \text{base address of array} + \\
\quad (\text{row} \times \text{size of row} \times \text{size of element}) + \\
\quad (\text{col} \times \text{size of element})
\]

-or-

\[
\text{base address of array} + \\
\quad (\text{row} \times \text{size of row} + \text{col} \times \text{size of element})
\]

In C, to define a 4x4 array of integers:

```c
int twodarr[4][4] = {{0x1, 0x2, 0x3, 0x4},
                      {0x4, 0x6, 0x7, 0x8},
                      {0x9, 0x10,0x11,0x12},
                      {0x13,0x14,0x15,0x16}};
```

The equivalent in X86 is:

```
.data
twodarr:    .long 0x1, 0x2, 0x3, 0x4
            .long 0x5, 0x6, 0x7, 0x8
            .long 0x9, 0x10,0x11,0x12
            .long 0x13,0x14,0x15,0x16
```

Either would be displayed using `gdb` as:

```
0x80497a0 <twodarr >: 0x000000001 0x000000002 0x000000003 0x000000004
0x80497b0 <twodarr+16>: 0x000000005 0x000000006 0x000000007 0x000000008
0x80497c0 <twodarr+32>: 0x000000009 0x000000010 0x000000011 0x000000012
0x80497d0 <twodarr+48>: 0x000000013 0x000000014 0x000000015 0x000000016
```