Arrays

Edited from the work of Ray Seyfarth

September 18, 2015
Arrays

- An array is a contiguous collection of memory cells of a specific type.
- The start address of an array is the address of the first element.
  - The start address is associated with the label given before a data definition in the data segment or a data reservation in the bss segment.
  - Unless the array is in allocated memory.
- The first index of an array in C/C++ and assembly is 0.
- Some high level languages use different or user-selectable starting indices for arrays.
  - Fortran defaults to 1.
Array address computation

- Array elements all have the same size: 1, 2, 4 and 8 are common
  - If I have a pointer in my struct. The heap allocated memory does not contribute to the size of the object.
  - The pointer itself contributes to the size of the object not the heap allocated memory
  - A pointer on 64bit linux. Is 64 bits.

- Suppose an array has elements of size 4 and starts at address 0x10000
  - The first element (at index 0) is at 0x10000
  - The second element (at index 1) is at 0x10004
  - The third element (at index 2) is at 0x10008
  - Element number $k$ is at address $0x10000 + k*4$
General pattern for memory references

[label]  the value contained at label
[label+ind]  the value contained at the memory address obtained by adding the label and index register
[label+2*ind]  the value contained at the memory address obtained by adding the label and index register times 2
[label+4*ind]  the value contained at the memory address obtained by adding the label and index register times 4
[label+8*ind]  the value contained at the memory address obtained by adding the label and index register times 8
General pattern for memory references

Consider:

    segment .data
c: dq 4,1,5,2,7,8

Then,

    mov rax, [c];

moves 4 into rax. And

    mov rcx,2
    mov rax, [c+8*rcx];

moves 5 into rax.

What would the following misguided move load?

    mov rcx,1
    mov rax, [c+4*rcx];
General pattern for memory references

[reg]  the value contained at the memory address in the register
[reg+k\*ind]  the value contained at the memory address obtained by adding the register and index register times k
[label+reg+k\*ind]  the value contained at the memory address obtained by adding the label, the register and index register times k
[n+reg+k\*ind]  the value contained at the memory address obtained by adding n, the register and index register times k
General pattern for memory references

Consider:

```assembly
segment .data
a: dq 123
c: dq 4,1,5,2,7,8
```

Then

```assembly
lea rcx,[c]    ; or mov rcx, c
mov rax,[rcx]
```

will load 4 into rax. And

```assembly
lea rcx,[a]
mov rdi,4
mov rax,[8 +rcx + 8*rdi]
```
General pattern for memory references

Consider:

```
segment .data
a: dq 123
c: dq 4,1,5,2,7,8
```

Then

```
lea rcx,[c] ; or mov rcx, c
mov rax,[rcx]
```

will load 4 into rax. And

```
lea rcx,[a]
mov rdi,4
mov rax,[8 +rcx + 8*rdi]
```

will load 7 into rax.
Memory references

- For items in the data and bss segments we can use a label.
- For arrays passed into functions the address is passed in a register.
- Soon we will be allocating memory using `malloc`.
  - This address will typically be stored in memory.
  - Later to use the data, we must load the address from memory into a register.
  - Then we can use a register form of memory reference.
- The use of a number or a label is equivalent to the computer.
  - Both use the same instruction and place the number or label value into the same field of the instruction.
  - Using multipliers of 2, 4 or 8 are essentially “free” with index registers.
Copy dword array example

- In the function below the first parameter is the address of the first dword of a destination array (rdi)
- The second parameter is the address of the source array (rsi)
- The third parameter is the number of dwords to copy (rdx)
- It would generally be faster to use “rep movsd”

```
copy_array:
    xor ecx, ecx ; index=0
more:   mov eax, [rsi+4*rcx] ; move src[index] to temp
        mov [rdi+4*rcx], eax ; move to dst[index]
        inc rcx ; ++index
        cmp rcx, rdx
        jne more
    xor eax, eax
    ret
; if rdx=0 bad things happen
```

©Edited from the work of Ray Seyfarth
Allocating arrays

If we wish to directly allocate heap storage in assembler we have two options.

- We can make use of the \texttt{brk} and \texttt{sbrk} system calls which allow us a means of altering the heap boundary.
- Or the more modern approach using \texttt{mmap}.

In this course we will however make use of the C \texttt{malloc} function.

- If \texttt{malloc} is not fast enough, your time would be better served rewriting a version of \texttt{malloc} for your purposes (maybe in ASM) rather than using the system calls directly all the time.
Allocating arrays

- We will allocate arrays using the C `malloc` function
  ```c
  void *malloc ( long size );
  ```
- The parameter to `malloc` is the number of bytes to allocate
- `malloc` returns the address of the array or 0
- Data allocated should be freed
  ```c
  void free ( void *ptr );
  ```
Code to allocate an array

- The code below allocates an array of 1 billion bytes
- It saves the pointer to the new array in memory location named `pointer`

```asm
extern malloc
...
mov rdi, 1000000000
call malloc
mov [pointer], rax
```
Advantages for using allocated arrays

- The array will be the right size
- There are size limits of about 2 GB in the data and bss segments
- The assembler is very slow with large arrays and the program is large
- Assembling a program with a 2 GB array in the data segment took about 100 seconds
- The executable was over 2 GB
- Using malloc the program assembles in less than 1 second and the executable as about 10 KB
We present an application which creates an array
Fills the array with random data by calling \texttt{random}
Prints the array if the size is small
Determines the minimum value in the array.
Only the helper functions will be discussed in the lecture.
Creating an array

- This function allocates an array of double words
- The number of double words is the only parameter
- Note the use of a stack frame to avoid any problems of stack misalignment

; array = create ( size );
create:
push rbp
mov rbp, rsp
imul rdi, 4
call malloc
leave
ret
Filling the array with random numbers

call random
mov rdx, [rbx+rcx*4]
mov rdx, [rbx+rcx*4]
mov rax, [rbx+rcx*4]
mov [rbx+rcx*4], rax
inc rcx
cmp rcx, [rbx+rcx*4]
jl .more
leave
ret
Printing the array

;       print ( array, size );
print:
.array equ 0
.size equ 8
.i equ 16
push rbp
mov rbp, rsp
sub rsp, 32
mov [rsp+.array], rdi
mov [rsp+.size], rsi
xor ecx, ecx
mov [rsp+.i], rcx
Printing the array

segment .data
.format:
    db "%10d",0x0a,0
segment .text
.more
    lea rdi, [.format]
    mov rdx, [rsp+.array]
    mov rcx, [rsp+.i]
    mov esi, [rdx+rcx*4]
    mov [rsp+.i], rcx
    xor eax, eax
    call printf
    mov rcx, [rsp+.i]
    inc rcx
    mov [rsp+.i], rcx
    cmp rcx, [rsp+.size]
    jl .more
leave
ret
Finding the minimum value in the array

- This function calls no other function
- There is no need for a stack frame
- A conditional move is faster than branching

```assembly
; x = min ( a, size );
min:
    mov    eax, [rdi] ; start with a[0]
    mov    rcx, 1
    .more
    mov    r8d, [rdi+rcx*4] ; get a[i]
    cmp    r8d, eax
    cmovl  eax, r8d ; move if smaller
    inc    rcx
    cmp    rcx, rsi
    jl     .more
    ret
```
The first argument to `main` is the number of command line parameters.

The second argument is the address of an array of character pointers, each pointing to one of the parameters.

Below is a C program illustrating the use of command line parameters.

```c
#include <stdio.h>

int main ( int argc, char *argv[] )
{
    int i;
    for ( i = 0; i < argc; i++ ) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```
segment .data
format  db    "%s",0x0a,0
segment .text
global main ; let the linker know about main
extern printf ; resolve printf from libc
main: push rbp ; prepare stack frame for main
      mov  rbp, rsp
      sub  rsp, 16
      mov  rcx, rsi ; move argv to rcx
      mov  rsi, [rcx] ; get first argv string

start_loop:
      lea  rdi, [format]
      mov  [rsp], rcx ; save argv
      call printf
      mov  rcx, [rsp] ; restore argv
      add  rcx, 8 ; advance to next pointer in argv
      mov  rsi, [rcx] ; get next argv string
      cmp  rsi, 0          ; end with NULL pointer
      jnz  start_loop

end_loop: