Arrays

Ray Seyfarth

August 2, 2011

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
- This is associated with the label given before a data definition in the data segment or a data reservation in the bss segment
- The first index of an array in C/C++ and assembly is 0
- Each subsequent array cell is at a higher memory address
- The final index for an array of n elements is $n-1$
- Some high level languages use different or user-selectable starting indices for arrays
- Fortran defaults to 1
- 0 is the most logical first index because it simplifies array address computation

Outline

1. Array address computation
2. General pattern for memory references
3. Allocating arrays
4. Processing arrays
5. Command line parameter array

Scripts to link programs and to execute gdb

- `gdb` is aware of addresses of labels but not their types
- In yasm data definition or reservation is by size of data elements
- For items in the data segment we can infer the intended type from the code
- "`a dd 125`" is a pretty good clue that an integer
- "`b dd 1.5`" tells us that b is a float
- You can use scripts `yld` and `ygcc` which link programs using either `ld` or `gcc` and prepare files with `gdb` macros
- Then using `ygdb` to run `gdb` will use these macros to give `gdb` better type information
- With this set of scripts everything is an array

Array address computation

- Array elements all have the same size: 1, 2, 4 and 8 are common
- 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`
- Let's examine the arrays for program "`array.asm`" with `gdb` and `ygdb`

```
        segment .bss
a       resb      100
b       resd      100
        align     8
c       resq      100
```

General pattern for memory references

`[label]`	the value contained at label
`[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
`[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

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

Example using base registers and an index register

- In the function below the first parameter is the address of the first dword of a destination array
- The second parameter is the address of the source array
- The third parameter is the number of dwords to copy
- It would generally be faster to use "`rep movsd`"

```
        segment .text
        global  copy_array
copy_array:
        xor     ecx, ecx
more:   mov     eax, [rsi+4*rcx]
        mov     [rdi+4*rcx], eax
        add     rcx, 1
        test    rcx, rdx
        jne     more
        xor     eax, eax
        ret
```

Allocating arrays

- We will allocate arrays using the C `malloc` function

```
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, although this will happen when a program exits

```
    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`

```
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 bss segment took about 100 seconds
- The executable was over 2 GB
- Using `malloc` the program assembles in less than 1 second and the program as about 10 KB
- Modified to allocate 20 billion bytes the program executes in 3 milliseconds

Processing arrays

- We present an application which creates an array
- Fills the array with random data by calling `random`
- Prints the array if the size is small (up to 20 elements)
- Determines the minimum value in the array

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

```
fill:
.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
.more   mov     [rsp+.i], rcx
        call    random
        mov     rcx, [rsp+.i]
        mov     rdi, [rsp+.array]
        mov     [rdi+rcx*4], eax
        inc     rcx
        cmp     rcx, [rsp+.size]
        jl      .more
        leave
        ret
```

Local labels in yasm

- Labels beginning with a dot are local labels
- They are considered part of the previous normal label
- The `.more` label could be referenced as `fill.more` from outside the `fill` function
- The `fill` function keeps saving `rcx` on the stack and restoring `rcx` and `rdi` around the `random` call
- This could be easier to code using registers which are preserved across calls

Filling the array with random numbers (2)

```
fill:
.r12    equ     0
.r13    equ     8
.r14    equ     16
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     [rsp+.r12], r12
        mov     [rsp+.r13], r13
        mov     [rsp+.r14], r14
        mov     r12, rdi        ; r12 is the array address
        mov     r13, rsi        ; r13 is the size
        xor     r14d, r14d      ; loop counter
.more   call    random
        mov     [r12+r14*4], eax
        inc     r14
        cmp     r14, r13
        jl      .more
        mov     r12, [rsp+.r12]
        mov     r13, [rsp+.r13]
        mov     r14, [rsp+.r14]
        leave
        ret
```


Printing the array

```
print:
.array  equu    0
.size   equ     8
.i      equ     16
        ...
        segment .data
.format:
        db      "%10d",0x0a,0
        segment .text
.more   lea     rdi, [.format]
        mov     rdx, [rsp+.array]
        mov     rcx, [rsp+.i]
        mov     rsi, [rdx+rcx*4]
        mov     [rsp+.i], rcx
        call    printf
        mov     rcx, [rsp+.i]
        inc     rcx
        mov     [rsp+.i], rcx
        cmp     rcx, [rsp+.size]
        jl      .more
```

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

```
;       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 main program and testing

- The code is too long, so we will inspect it in an editor
- It's also time to test with `gdb`

Command line parameter array

- 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

```
#include <stdio.h>

int main ( int argc, char *argv[] )
{
    int i;
    for ( i = 0; i < argc; i++ ) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Assembly program listing command line parameters

```
        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 rsi
        add     rcx, 8       ; advance to next pointer in argv
        mov     rsi, [rcx]   ; get next argv string
        cmp     rsi, 0
        jnz     start_loop   ; end with NULL pointer
end_loop:
```