COMP 3221

Microprocessors and Embedded Systems

Lectures 18 : Pointers & Arrays in C/ Assembly

http://www.cse.unsw.edu.au/~cs3221

September, 2003

Saeid Nooshabadi

Saeid@unsw.edu.au

Overview

- Arrays, Pointers, Functions in C
- Example
- Pointers, Arithmetic, and Dereference
- Conclusion

Review: Register Convention

- Caller Saved Registers:
  - Return address `lr`
  - Arguments `a1, a2, a3, a4`
  - Return values `a1, a2, a3, a4`
- Callee Saved Registers:
  - `v` Registers `v1 – v8`

Review: Function Call Bookkeeping

- Big Ideas:
  - Follow the procedure conventions and nobody gets hurt.
  - Data is just 1's and 0's, what it represents depends on what you do with it
- Function Call Bookkeeping:
  - Caller Saved Registers are saved by the caller, that is, the function that includes the `bl` instruction
  - Callee Saved Registers are saved by the callee, that is, the function that includes the `mov pc, lr` instruction
  - Some functions are both a caller and a callee

Argument Passing Options

- 2 choices
  - "Call by Value": pass a copy of the item to the function/procedure
    - `x` … `f(x)` … `x`. Call to `f` does not change `x`
  - "Call by Reference": pass a pointer to the item to the function/procedure
- Single word variables passed by value
- What about passing an array? e.g., `a[100]`
  - Pascal--call by value--copies 100 words of `a[]` onto the stack
  - C--call by reference--passes a pointer (1 word) to the array `a[]` in a register

Pointers Implementation in ARM

- `c` is `int`, has value 100, in memory at address 0x10000000, `p` in `v1`, `x` in `v2`

```
 p = &c;  /* p gets 0x10000000 */
 x = *p;  /* x gets 100 */
*p = 200; /* c gets 200 */
```

```
 ; p = &c;   /* p gets 0x10000000 */
mov v1,0x1000000 ; p = 0x10000000

 ; x = *p;   /* x gets 100 */
ldr  v2, [v1]    ; dereferencing p

; *p = 200; /* c gets 200 */
mov  a1, #200
str  a1, [v1]    ; dereferencing p
```


Simple Array: C vs. ARM Assembly

```
int strlen(char *s) {
 char *p = s; /* p points to chars */

  while (*p != '\0')
   p++; /* points to next char */
 return p - s; /* end - start */
}
```

```
      sub  a2, a1, #1; p = s - 1
 Loop: ldrb a3,[a2,#1]!
                     ;/*derefence p, p++*/
      cmp a3, #0
      bne Loop

Exit:  sub a1,a2,a1 ; p - s
      mov pc, lr
```


Arrays, Pointers, Functions in C

- 4 versions of array function that adds two arrays and puts sum in a third array (`sumarray`)
  - Third array is passed to function
  - Using a local array (on stack) for result and passing a pointer to it
  - Third array is allocated on heap
  - Third array is declared static
- Purpose of example is to show interaction of C statements, pointers, and memory allocation

Calling `sumarray`, Version 1

```
int x[100], y[100], z[100];

sumarray(x, y, z);
```

° C calling convention means above the same as

```
sumarray(&x[0], &y[0], &z[0]);
```

° Really passing pointers to arrays

```
mov   a1,sb          ; x[0] starts at sb
add   a2,sb,#400     ; y[0] above x[100]
add   a3,sb,#800     ; z[0] above y[100]
bl    sumarray
```


Version 1: Optimized Compiled Code

```
void sumarray(int a[],int b[],int c[]) {
 int i;

 for(i=0;i<100;i=i+1)
     c[i] = a[i] + b[i];
}
sumarray: stmfd sp!,{v1- v2};save v1-v2 on stack
          add a4, a1,#400     ; beyond end of a[]
Loop:     cmp a1, a4
          beq Exit
          ldr v1, [a1], #4 ;a1=a[i], a1=a1+4
          ldr v2, [a2], #4 ;a2=b[i], a2=a2+4
          add v2, v2, v1   ;v2=a[i] + b[i]
          str v2, [a3], #4 ;c[i]=a[i] + b[i]
                           ; a3 = a3+4
          b Loop
Exit:     ldmfd sp!,{v1-v2}; restore v1-v2
          mov pc, lr
```


Version 2 to Fix Weakness of Version 1

° Would like recursion to work

```
int * sumarray(int a[],int b[]);
 /* adds 2 arrays and returns sum */

sumarray(x, sumarray(y,z));
```

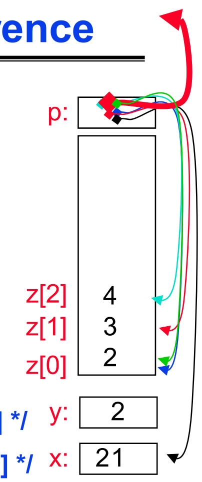
° Cannot do this with Version 1 style solution: what about this

```
int * sumarray(int a[],int b[]) {
    int i, c[100];
    for(i=0;i<100;i=i+1)
         c[i] = a[i] + b[i];
    return c;
}
```


Pointers, Arithmetic, and Dereference

```
int x = 1,  y = 2;   /* x and y are integer variables */
int z[10];           /* an array of 10 ints, z points to start */
int *p;              /* p is a pointer to an int */

x = 21;              /* assigns x the new value 21 */
z[0] = 2; z[1] = 3   /* assigns 2 to the first, 3 to the next */
p = &z[0];           /* p refers to the first element of z */
p = z;               /* same thing;  p[ i ] == z[ i ]*/
p = p+1;             /* now it points to the next element, z[1] */
p++;                 /* now it points to the one after that, z[2] */
*p = 4;              /* assigns 4 to there, z[2] == 4*/
p = 3;               /* bad idea! Absolute address!!! */
p = &x;              /* p points to x, *p == 21  */
z = &y               illegal!!!!!    array name is not a variable
```



Version 2: Revised Compiled Code

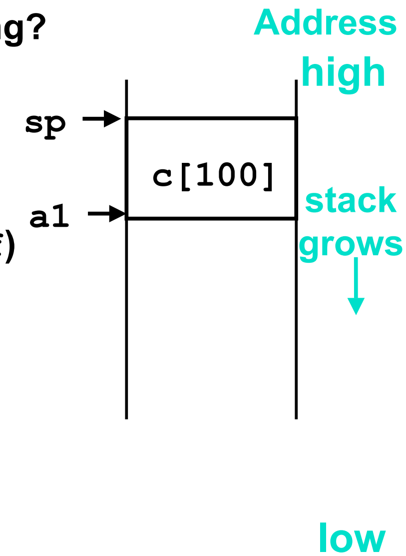
```
 for(i=0;i<100;i=i+1)
     c[i] = a[i] + b[i];
  return c;}

sumarray: stmfd sp!,{v1- v2};save v1-v2 on stack
          add a4, a1,#400      ; beyond end of a[]
          sub sp, sp,#400      ; space for c
          mov a3, sp           ; ptr for c
Loop:     cmp a1, a4
          beq Exit
          ldr v1, [a1], #4 ;a1=a[i], a1=a1+4
          ldr v2, [a2], #4 ;a2=b[i], a2=a2+4
          add v2, v2, v1   ;v2=a[i] + b[i]
          str v2, [a3], #4 ;c[i]=a[i] + b[i]
                               ; a3 = a3+4

          b Loop
Exit:     mov  a1, sp          ; &c[0]
          add  sp,sp, 400      ; pop stack
          ldmfd sp!,{v1-v2}; restore v1-v2
          mov pc, lr
```


Weakness of Version 2

- Legal Syntax; What's Wrong?
- Will work until call another function that uses stack
- `c[100]` Won't be reused instantly(e.g, add a `printf`)
- Stack allocated + unrestricted pointer is problem



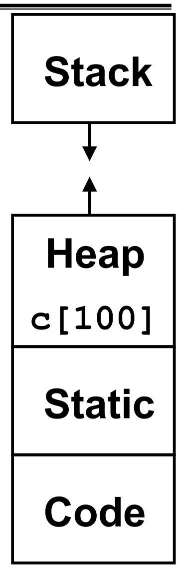
Version 3 to Fix Weakness of Version 2

- Solution: allocate `c[]` on heap

```
int * sumarray(int a[],int b[]) {
    int i;
    int *c;

    c = (int *) malloc(100);

    for(i=0;i<100;i=i+1)
          c[i] = a[i] + b[i];
    return c;
}
```



- Not reused unless freed
  - Can lead to memory leaks
  - Java, has garbage collectors to reclaim free space

Version 3: Revised Compiled Code

```
sumarray: stmfd sp!,{a1-a2,v1- v2,lr}
      ;save a1-a2, v1-v2 & lr on stack
          add a4, a1,#400      ; beyond end of a[]
          mov a1,#400          ;
          bl malloc            ; get space for c
          mov a3, a1           ; get &c
          ldmfd sp!,{a1-a2}    ; restor a1-a2
Loop:     cmp a1, a4
          beq Exit
          ldr v1, [a1], #4 ;a1=a[i], a1=a1+4
          ldr v2, [a2], #4 ;a2=b[i], a2=a2+4
          add v2, v2, v1   ;v2=a[i] + b[i]
          str v2, [a3], #4 ;c[i]=a[i] + b[i]
                           ; a3 = a3+4
          b Loop
Exit:     sub  a1, a3, #400    ; &c[0]
          ldmfd sp!,{v1-v2,pc}; restore v1-v2
                               ; and return
```

Lifetime of storage & scope

- automatic (stack allocated)
  - typical local variables of a function
  - created upon call, released upon return
  - scope is the function
- heap allocated
  - created upon `malloc`, released upon `free`
  - referenced via pointers
- external / static
  - exist for entire program

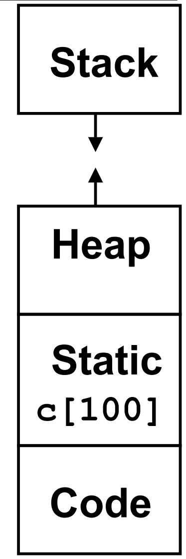
Version 4 : Alternative to Version 3

- Static declaration

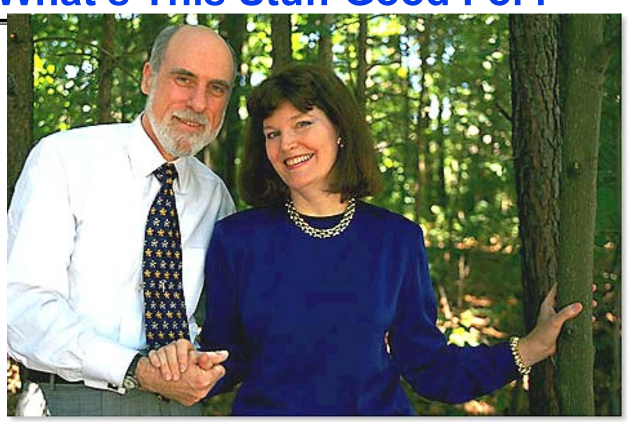
```
int * sumarray(int a[],int b[]) {
    int i;
    static int c[100];

    for(i=0;i<100;i=i+1)
           c[i] = a[i] + b[i];
    return c;
}
```

- Compiler allocates once for function, space is reused
  - Will be changed next time `sumarray` invoked
  - Used in C libraries



"What's This Stuff Good For?"



In 1974 Vint Cerf co-wrote TCP/IP, the language that allows computers to communicate with one another. His wife of 35 years (Sigrid), hearing-impaired since childhood, began using the Internet in the early 1990s to research cochlear implants, electronic devices that work with the ear's own physiology to enable hearing. Unlike hearing aids, which amplify all sounds equally, cochlear implants allow users to clearly distinguish voices--even to converse on the phone. Thanks in part to information she gleaned from a chat room called "Beyond Hearing," Sigrid decided to go ahead with the implants in 1996. The moment she came out of the operation, she immediately called home from the doctor's office--a phone conversation that Vint still relates with tears in his eyes. ***One Digital Day*, 1998 (www.intel.com/onedigitalday)**

What about Structures?

- Scalars passed by value
- Arrays passed by reference (pointers)
- Structures by value too
- Can think of C passing everything by value, just that arrays are simply a notation for pointers and the pointer is passed by value

"And in Conclusion.."

° **In C :**
  • **Scalars passed by value**
  • **Arrays passed by reference**

° **In C functions we can return a pointer to Arrays defined in Static, Heap or stack area.**

° **Returning a pointer to an array in stack gives rise to unrestricted pointers**