CS61C - Machine Structures

Lecture 8 - Procedure Conventions part II, plus, Arrays and Pointers in C

September 22, 2000

Sumeet Shendrikar& Daniel C. Silverstein

http://www-inst.eecs.berkeley.edu/~cs61c/

Review 1/3

° 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 jal instruction
- Callee Saved Registers are saved by the callee, that is, the function that includes the jr $ra instruction
- Some functions are both a caller and a callee

Review 2/3

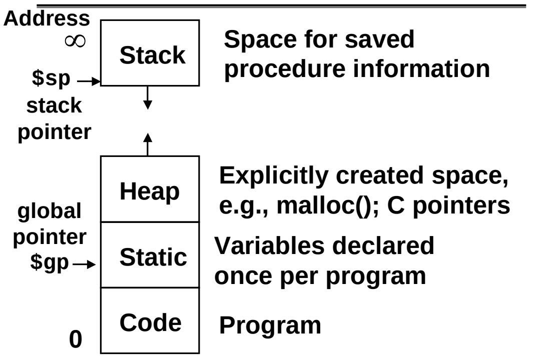
• Caller Saved Registers:

• Return address	$ra
• Arguments	$a0, $a1, $a2, $a3
• Return value	$v0, $v1
• $t Registers	$t0 - $t9

• Callee Saved Registers:

• $s Registers	$s0 - $s7

Review 3/3



Space for saved procedure information

Explicitly created space, e.g., malloc(); C pointers

Variables declared once per program

Program

Overview

° Why Procedure Conventions?

° Basic Structure of a Function

° Example: Recursive Function

° Administrivia

° Arrays, Pointers, Functions in C

° Example

° Pointers, Arithmetic, and Dereference

° Conclusion

Why Procedure Conventions? 1/2

° Think of procedure conventions as a contract between the Caller and the Callee
- If both parties abide by a contract, everyone is happy (:))
- If either party breaks a contract, disaster and litigation result (: O)

° Similarly, if the Caller and Callee obey the procedure conventions, there are significant benefits. If they don't, disaster and program crashes result

Why Procedure Conventions? 2/2

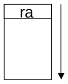
- Benefits of Obeying Procedure Conventions:
  - People who have never seen or even communicated with each other can write functions that work together
  - Recursion functions work correctly

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)# save $ra
save other regs
```

Body `...`



Epilogue

```
restore other regs
lw $ra, framesize-4($sp)# restore $ra
addi $sp,$sp, framesize
jr $ra
```

Example: Fibonacci Numbers 1/

- The Fibonacci numbers are defined as follows: F(n) = F(n – 1) + F(n – 2), F(0) and F(1) are defined to be 1
- In scheme, this could be written:

```
(define (Fib n)
 (cond   ((= n 0) 1)
         ((= n 1) 1)
         (else (+ (Fib (- n 1))
                  (Fib (- n 2)))))
```

Example: Fibonacci Numbers 2/

- Rewriting this in C we have:

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

Example: Fibonacci Numbers 3/

- Now, let's translate this to MIPS!
- You will need space for three words on the stack
- The function will use one $s register, $s0
- Write the Prologue:

```
fib:
addi $sp, $sp, -12     # Space for three words
sw $ra, 8($sp)         # Save the return address
sw $s0, 4($sp)         # Save $s0
```

Example: Fibonacci Numbers 4/

- Now write the Epilogue:

```
fin:
lw $s0, 4($sp)         # Restore $s0
lw $ra, 8($sp)         # Restore return address
addi $sp, $sp, 12      # Pop the stack frame
jr $ra                 # Return to caller
```

Example: Fibonacci Numbers 5/

° Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
 if(n == 0) { return 1; } /*Translate Me!*/
 if(n == 1) { return 1; } /*Translate Me!*/
 return (fib(n - 1) + fib(n - 2));
}
```

```
addi $v0, $zero, 1      # $v0 = 1
beq $a0, $zero, fin     # if (n == 0). . .
addi $t0, $zero, 1      # $t0 = 1
beq $a0, $t0, fin       # if (n == 1). . .
Contiued on next slide. . .
```

Example: Fibonacci Numbers 6/8

° Almost there, but be careful, this part is tricky!

```
int fib(int n) {
 . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
addi $a0, $a0, -1       # $a0 = n - 1
sw $a0, 0($sp)          # Need $a0 after jal
jal fib                 # fib(n – 1)
lw $a0, 0($sp)          # Restore $a0
addi $a0, $a0, -1       # $a0 = n – 2
Continued on next slide. . .
```

Example: Fibonacci Numbers 7/8

° Remember that $v0 is caller saved!

```
int fib(int n) {
 . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
add $s0, $v0, $zero     # Place fib(n – 1)
                        # somewhere it won’t get
                        # clobbered
jal fib                 # fib(n – 2)
add $v0, $v0, $s0       # $v0 = fib(n-1) + fib(n-2)
To the epilogue and beyond. . .
```

Example: Fibonacci Numbers 8/8

° Here’s the complete code for reference:

```
fib:
addi $sp, $sp, -12
sw $ra, 8($sp)
sw $s0, 4($sp)
addi $v0, $zero, 1
beq $a0, $zero, fin
addi $t0, $zero, 1
beq $a0, $t0, fin
addi $a0, $a0, -1
sw $a0, 0($sp)
jal fib
lw $a0, 0($sp)
addi $a0, $a0, -1
add $s0, $v0, $zero
jal fib
add $v0, $v0, $s0
fin:
lw $s0, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
jr $ra
```

Administrivia 1/2

- Most assignments are now submitted online (even homeworks)!
- Proj2 (sprintf) due date moved to Sunday (9/24) at midnight
  - You voted on this in Wed lecture
  - TAs will NOT be in the labs on Sat/Sun so seek help NOW if you need it.

° Remember that you must use proper register/proc conventions in Proj2

- Lab4 due by the beginning of your discussion this week

Administrivia 2/2

- M’Piero has found Professor Patterson!
- He’ll be back next week

Argument Passing Options

- 2 choices
  - "<u>Call by Value</u>": pass a <u>copy</u> of the item to the function/procedure
  - "<u>Call by Reference</u>": pass a <u>pointer</u> to the item to the function/procedure
- Single word variables passed by value
- 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

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

```
addi $a0,$gp,0   # x[0] starts at $gp
addi $a1,$gp,400 # y[0] above x[100]
addi $a2,$gp,800 # z[0] above y[100]
jal  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];

}
     addi $t0,$a0,400 # beyond end of a[]
Loop:beq  $a0,$t0,Exit
     lw   $t1, 0($a0) # $t1=a[i]
     lw   $t2, 0($a1) # $t2=b[i]
     add  $t1,$t1,$t2 # $t1=a[i] + b[i]
     sw   $t1, 0($a2) # c[i]=a[i] + b[i]
     addi $a0,$a0,4   # $a0++
     addi $a1,$a1,4   # $a1++
     addi $a2,$a2,4   # $a2++
     j    Loop
Exit:jr   $ra
```

Version 2 to Fix Weakness of Version 1

- Would like recursion to work

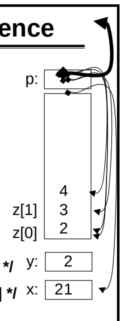
```
int sumarray(int a[],int b[]);
 /* adds 2 arrays and returns sum */

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

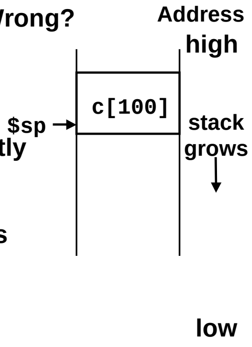


Version 2: Revised Compiled Code

```
 for(i=0;i<100;i=i+1)
    c[i] = a[i] + b[i];
 return c;}
     addi $t0,$a0,400 # beyond end of a[]
     addi $sp,$sp,-400# space for c
     addi $t3,$sp,0   # ptr for c
     addi $v0,$t3,0   # $v0 = &c[0]
Loop:beq  $a0,$t0,Exit
     lw   $t1, 0($a0) # $t1=a[i]
     lw   $t2, 0($a1) # $t2=b[i]
     add  $t1,$t1,$t2 # $t1=a[i] + b[i]
     sw   $t1, 0($t3) # c[i]=a[i] + b[i]
     addi $a0,$a0,4   # $a0++
     addi $a1,$a1,4   # $a1++
     addi $t3,$t3,4   # $t3++
     j    Loop
Exit:addi $sp,$sp, 400# pop stack
     jr   $ra
```

Weakness of Version 2

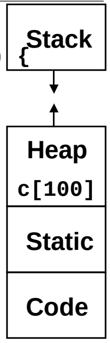
- Legal Syntax; What's Wrong?
- Will work until call another function that uses stack
- Won't be reused instantly (e.g, add a `printf`)
- Stack allocated + unrestricted pointer is problem



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, Scheme have garbage collectors to reclaim free space

Version 3: Revised Compiled Code

```
     addi $t0,$a0,400 # beyond end of a[]
     addi $sp,$sp,-12 # space for regs
     sw   $ra, 0($sp) # save $ra
     sw   $a0, 4($sp) # save 1st arg.
     sw   $a1, 8($sp) # save 2nd arg.
     addi $a0,$zero,400 #
     jal  malloc
     addi $t3,$v0,0   # ptr for c
     lw   $a0, 4($sp) # restore 1st arg.
     lw   $a1, 8($sp) # restore 2nd arg.
Loop:beq  $a0,$t0,Exit
     ... (loop as before on prior slide )
     j    Loop
Exit:lw   $ra, 0($sp) # restore $ra
     addi $sp,$sp, 12 # pop stack
     jr   $ra
```

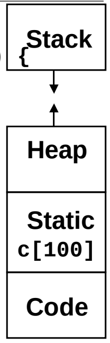
Lifetime of storage & scope

- 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

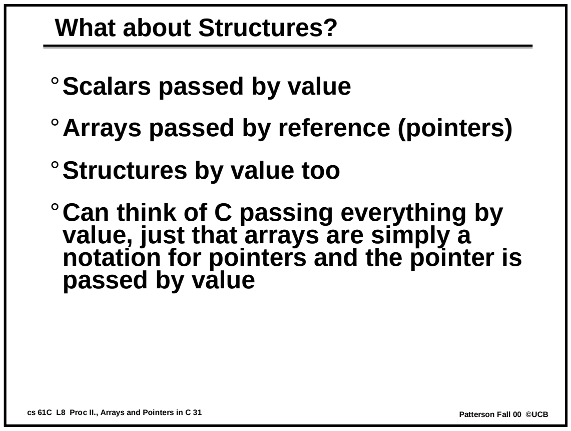
Version 4 : Alternative to Version 3

- 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;
}
```

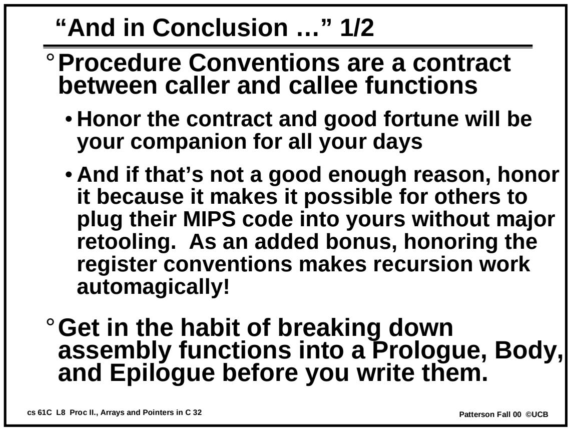


- Compiler allocates once for function, space is reused
  - Will be changed next time `sumarray` invoked
  - Why describe? used in C libraries



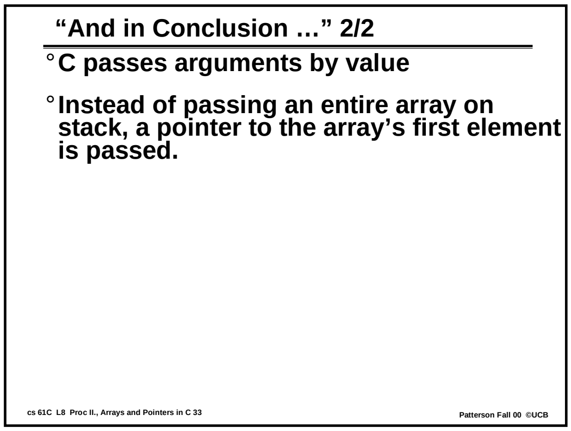
What about Structures?

- 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 …" 1/2

- Procedure Conventions are a contract between caller and callee functions
  - Honor the contract and good fortune will be your companion for all your days
  - And if that's not a good enough reason, honor it because it makes it possible for others to plug their MIPS code into yours without major retooling. As an added bonus, honoring the register conventions makes recursion work automagically!
- Get in the habit of breaking down assembly functions into a Prologue, Body, and Epilogue before you write them.



"And in Conclusion …" 2/2

- C passes arguments by value
- Instead of passing an entire array on stack, a pointer to the array's first element is passed.