Machine Level Programming: Recursion and Memory Layout of Arrays

CSCI 237: Computer Organization
18th Lecture, Monday, Oct. 21

Kelly Shaw

1

1

2

2

Administrative Details

- Read CSAPP 3.8, 3.9-3.10.4,
- Midterm in lab on Wednesday 10/23
  - Closed book and closed notes
  - Review session today 7-8:30pm in Physics 205
  - Sample midterm and solutions on Glow
- Snack and Gab on Wednesday 4:10-4:30 in CS Commons

3

3

Last Time: Machine-Level Programming: Procedures and Arrays

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

4

4

Today: Machine-Level Programming: Procedures and Arrays

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion
- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

5

5

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

6

6

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

7

7

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument

. . .
Rtn address
Saved %rbx ← %rsp

8

8

Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rdi`	`x >> 1`	Rec. argument
`%rbx`	`x & 1`	Callee-saved

9

9

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Recursive call return value	

10

10

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Return value	

11

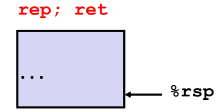
11

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
`%rax`	Return value	Return value



12

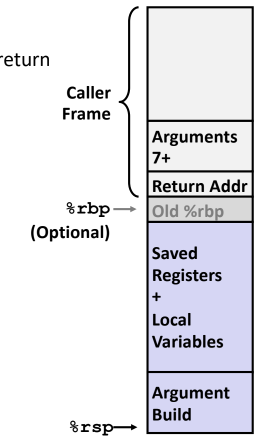
12

Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has “private” storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

Ch 3.7 Summary

- Important Points
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- Pointers are addresses of values



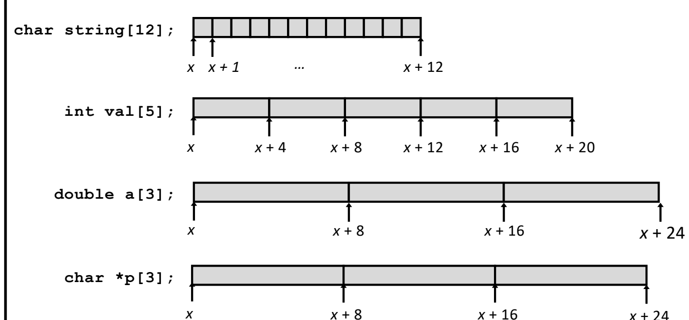
Today: Machine-Level Programming: Procedures and Arrays

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion
- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

Ch 3.8 - Array Allocation

- Basic Principle

  T **a**[*L*];
  - Array **a** of data type *T* and length *L*
  - **Contiguously** allocated region of *L* * `sizeof` (*T*) bytes in memory



Array Access

- Basic Principle
  T **a**[*L*];
  - Array **a** of data type *T* and length *L*
  - Identifier **a** can be used as a pointer to array element 0: Type *T**

`int val[5];`

	0	1	2	6	7
Address:	x	x + 4	x + 8	x + 12	x + 16	x + 20

Reference	Type	Value	
`val[4]`	`int`	`7`	
`val`	`int *`	`x`	
`val+1`	`int *`	`x + 4`	
`&val[2]`	`int *`	`x + 8`	
`val[5]`	`int`	`??`	
`*(val+1)`	`int`	`1`	`// val[1]`
`val + i`	`int *`	`x + 4 * i`	`//&val[i]`

17

17

Array Example

```
#define ZLEN 5
typedef int eph_val[ZLEN];

eph_val cow = { 0, 1, 2, 6, 7 };
eph_val eph = { 9, 0, 2, 1, 0 };
eph_val moo = { 8, 6, 7, 5, 3 };
```

`eph_val cow;`

0	1	2	6	7	
16	20	24	28	32	36

`eph_val eph;`

9	0	2	1	0	
36	40	44	48	52	56

`eph_val moo;`

8	6	7	5	3	
56	60	64	68	72	76

- Declaration "`eph_val cow`" equivalent to "`int cow[5]`"
- These example arrays were allocated in *successive* 20 byte blocks
  - Not guaranteed to happen in general.

18

18

Array Accessing Example

`eph_val cow;`

0	1	2	6	7	
16	20	24	28	32	36

```
int get_value
  (eph_val z, int index)
{
  return z[index];
}
```

x86-64

```
  # %rdi = z
  # %rsi = index
movl  (%rdi,%rsi,4), %eax   # z[index]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired value at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`
- `D(base, index, S)` `base+ D + index*S`

19

19

Array Loop Example

```
void zincr(eph_val z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax
  jmp     .L3
.L4:
  addl    $1, (%rdi,%rax,4)
  addq    $1, %rax
.L3:
  cmpq    $4, %rax
  jbe     .L4
  rep; ret
```

20

20

Practice on Your Own

- Convert the following x86-64 assembly code to C.

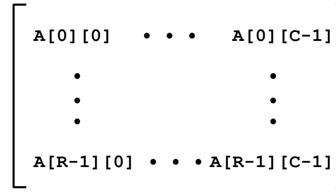
```
fcn:
    movl $0, %eax
    testq %rdi, %rdi
    je .L1
    movq (%rdi), %r8
    testq %r8, %r8
    je .L1
    pushq %r8
    addq $8, %rdi
    callq fcn
    popq %r8
    addq %r8, %rax
.L1:
    retq
```

21

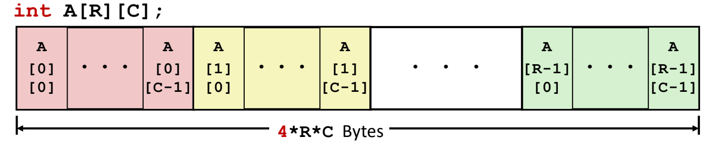
21

Multidimensional (Nested) Arrays

- Declaration
  T **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- Array Size
  - *R* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering



`int A[R][C];`

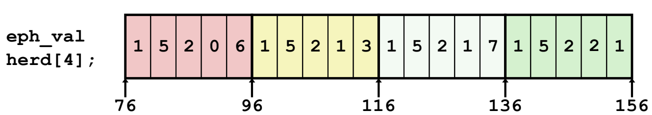


22

22

Nested Array Example

```
#define PCOUNT 4
eph_val herd[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



- "`eph_val herd[4]`" equivalent to "`int herd[4][5]`"
  - Variable **`herd`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

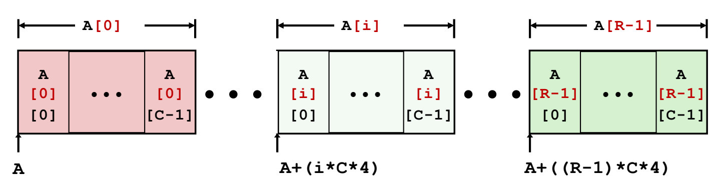
23

23

Nested Array Row Access

- Row Vectors
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address: **`A + i * (C * K)`**

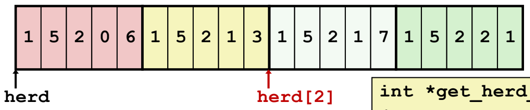
`int A[R][C];`



24

24

Nested Array Row Access Code




```
int *get_herd_eph(int index)
{
  return herd[index];
}
```

```
 # %rdi = index
  leaq (%rdi,%rdi,4),%rax # 5 * index
  leaq herd(,%rax,4),%rax # herd + (20 * index)
```

(address or label)

- Row Vector
  - `herd[index]` is array of 5 `int`'s
  - Starting address `herd + 20*index`
- Machine Code
  - Computes and returns address
  - Compute as `herd + 4*(index+4*index)`

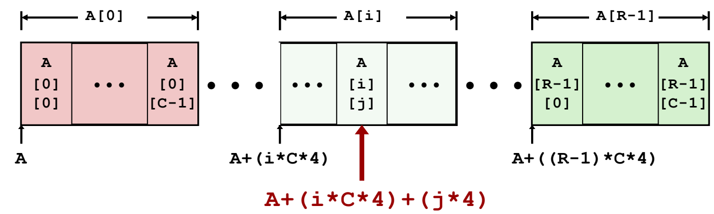
25

25

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - Address `A + i * (C * K) +  j * K`
    `= A + (i * C) * K + j * K`
    `= A + (i * C + j) * K`

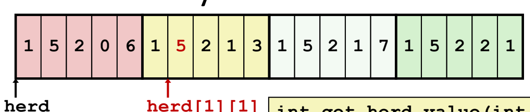
`int A[R][C];`



26

26

Nested Array Element Access Code




```
int get_herd_value(int index, int val)
{
  return herd[index][val];
}
```

```
 leaq   (%rdi,%rdi,4), %rax    # 5*index
 addl   %rax, %rsi             # 5*index+val
 movl   herd(,%rsi,4), %eax    # M[herd + 4*(5*index+val)]
```

- Array Elements
  - `herd[index][val]` is `int`
  - Address: `herd + 20*index + 4*val`
    `= herd + 4*(5*index + val)`

27

27