

Last Time

- **Stack**
 - Grows downwards in the address space
 - Main purpose is to support procedure call / return
- **Stack frames**
 - One per call to a procedure
 - Frame pointer (%EBP) points to bottom of frame
 - Stack pointer (%ESP) points to top of frame
 - Recursion means multiple stack frames corresponding to the same procedure might exist at the same time
- **Calling convention**
 - How are arguments passed?
 - Where does return value go?
 - Which registers are saved by caller / callee?

1

Today: Machine-Level Programming IV: Structured Data

- **Topics**
 - Arrays
 - Structs
 - Unions

2

Basic Data Types

- **Integer**
 - Stored and operated on using general-purpose registers
 - Signed vs. unsigned depends on instructions used

Intel	gas	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
- **Floating Point**
 - Stored and operated on in floating point registers

Intel	gas	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

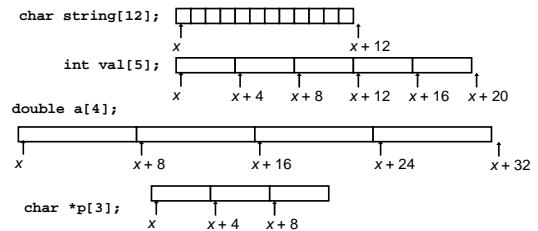
3

Array Allocation

- **Basic Principle**

```
T A[L];
```

 - Array of data type T and length L
 - Contiguously allocated region of $L * \text{sizeof}(T)$ bytes



4

Array Access

- **Basic Principle**

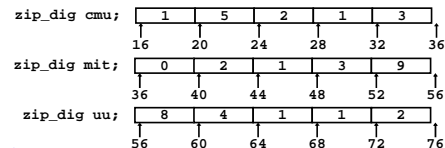
```
T A[L];
```

 - Array of data type T and length L
 - Identifier A can be used as a pointer to array element 0
-
- `int val[5];` 1 5 2 1 3
 x $x+4$ $x+8$ $x+12$ $x+16$ $x+20$
- **Reference**
- | | Type | Value |
|--------------------------|-------|--------|
| <code>val[4]</code> | int | 3 |
| <code>val</code> | int * | x |
| <code>val+1</code> | int * | $x+4$ |
| <code>&val[2]</code> | int * | $x+8$ |
| <code>val[5]</code> | int | ?? |
| <code>*(val+1)</code> | int | 5 |
| <code>val + i</code> | int * | $x+4i$ |

5

Array Example

```
typedef int zip_dig[5];
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig uu = { 8, 4, 1, 1, 2 };
```



- **Notes**
 - Declaration "`zip_dig uu`" equivalent to "`int uu[5]`"
 - Example arrays were allocated in successive 20 byte blocks
 - » Not guaranteed to happen in general

6

Array Accessing Example

Computation

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at $4 * \%eax + \%edx$
- Use memory reference (`%edx, %eax, 4`)

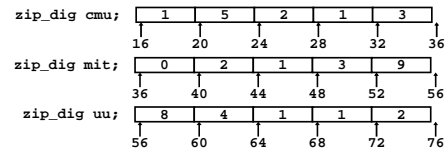
```
int get_digit (zip_dig z,
              int dig)
{
    return z[dig];
}
```

Memory Reference Code

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4), %eax # z[dig]
```

7

Referencing Examples



reference

- | | address | value | guaranteed? |
|----------------------|--------------------|-------|-------------|
| <code>mit[3]</code> | $36 + 4 * 3 = 48$ | 3 | Yes |
| <code>mit[5]</code> | $36 + 4 * 5 = 56$ | 8 | No |
| <code>mit[-1]</code> | $36 + 4 * -1 = 32$ | 3 | No |
| <code>cmu[15]</code> | $16 + 4 * 15 = 76$ | ?? | No |
- Out of range behavior implementation-dependent
 - No guaranteed relative allocation of different arrays
- No bounds checking is provided

8

Array Loop Example

Original Source

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

Transformed Version

- As generated by GCC
- Eliminate loop variable `i`
- Convert array code to pointer code
- Express in do-while form
 - No need to test at entrance

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```

9

Array Loop Implementation

Registers

```
%ecx z
%eax zi
%ebx zend
```

Computations

- $10 * zi + *z$ implemented as $*z + 2 * (zi + 4 * zi)$
- `z++` increments by 4

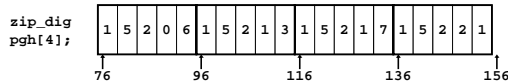
```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```

```
# %ecx = z
xorl %eax,%eax # zi = 0
leal 16(%ecx),%ebx # zend = z+4
.L59:
leal (%eax,%eax,4),%edx # 5*zi
movl (%ecx),%eax # *z
addl $4,%ecx # z++
leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
cmpl %ebx,%ecx # z : zend
jle .L59 # if <= goto loop
```

10

Nested Array Example

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



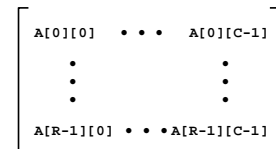
- Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
 - Variable `pgh` denotes array of 4 elements
 - Allocated contiguously
 - Each element is an array of 5 ints
 - Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

11

Nested Array Allocation

Declaration

- `T A[R][C]`;
 - Array of data type `T`
 - `R` rows, `C` columns
 - Type `T` element requires `K` bytes

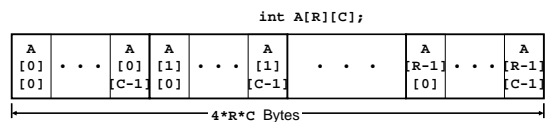


Array Size

- $R * C * K$ bytes

Arrangement

- Row-Major Ordering



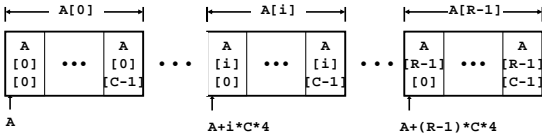
12

Nested Array Row Access

- **Row Vectors**

- $A[i]$ is array of C elements
- Each element of type T
- Starting address $A + i * C * K$

```
int A[R][C];
```



13

Nested Array Row Access Code

```
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

- **Row Vector**

- $pgh[index]$ is array of 5 int's
- Starting address $pgh + 20 * index$

- **Code**

- Computes and returns address
- Compute as $pgh + 4 * (index + 4 * index)$

```
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(,%eax,4),%eax # pgh + (20 * index)
```

14

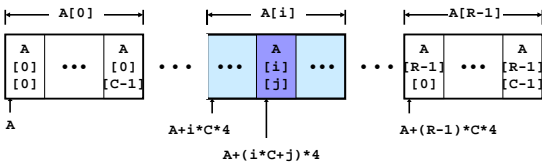
Nested Array Element Access

- **Array Elements**

- $A[i][j]$ is element of type T
- Address $A + (i * C + j) * K$



```
int A[R][C];
```



15

Nested Array Element Access Code

- **Array Elements**

- $pgh[index][dig]$ has type int
- Address: $pgh + 20 * index + 4 * dig$

```
int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```

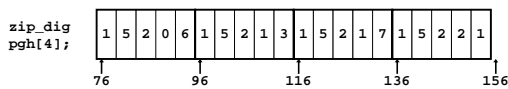
- **Code**

- Computes address
- $pgh + 4 * dig + 4 * (index + 4 * index)$
- `movl` performs memory reference

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx # 4*dig
leal (%eax,%eax,4),%eax # 5*index
movl pgh(%edx,%eax,4),%eax # *(pgh + 4*dig + 20*index)
```

16

Strange Referencing Examples



	address	value	reference guaranteed?
<code>pgh[3][3]</code>	$76 + 20 * 3 + 4 * 3 = 148$	2	Yes
<code>pgh[2][5]</code>	$76 + 20 * 2 + 4 * 5 = 136$	1	Yes
<code>pgh[2][-1]</code>	$76 + 20 * 2 + 4 * -1 = 112$	3	Yes
<code>pgh[4][-1]</code>	$76 + 20 * 4 + 4 * -1 = 152$	1	Yes
<code>pgh[0][19]</code>	$76 + 20 * 0 + 4 * 19 = 152$	1	Yes
<code>pgh[0][-1]</code>	$76 + 20 * 0 + 4 * -1 = 72$??	No

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

17

Comment

- **Array memory ordering is language dependent**

- for C it's row-major order
 - › General case: subscripts in order of most to least significant effect on computed address
- for Fortran however it's column-major order
 - › General case: subscripts in order of least to most significant

- **Somewhat analogous to the Endian issues**

- incompatibility creates confusion and portability problems

- **However it's worse**

- C can call Fortran on the same machine!
 - › and if references are done by pointers then you'll need to keep it all straight
- Ouch

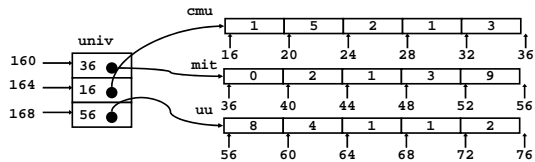
18

Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
 - > 4 bytes
- Each pointer points to array of int's

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig uu = { 8, 4, 1, 1, 2 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, uu};
```



19

Element Access in Multi-Level Array

```
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```

Computation

- Element access
 - $\text{Mem}[\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig}]$
- Must do two memory reads
 - > First get pointer to row array
 - > Then access element within array

```
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx # 4*index
movl univ(%edx),%edx # Mem[univ+4*index]
movl (%edx,%eax,4),%eax # Mem[...+4*dig]
```

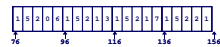
20

Array Element Accesses

- Similar C references
- Nested Array**

```
int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```

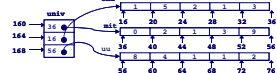
- Element at
 - $\text{Mem}[\text{pgh}+20*\text{index}+4*\text{dig}]$



- Different address computation
- Multi-Level Array**

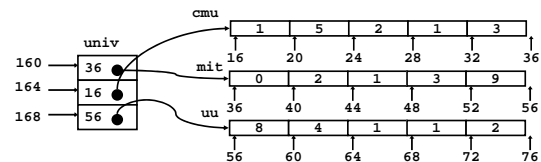
```
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```

- Element at
 - $\text{Mem}[\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig}]$



21

Strange Referencing Examples

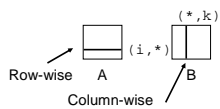


- | | address | value | reference guaranteed? |
|--------------------------|----------------|-------|-----------------------|
| <code>univ[2][3]</code> | $56+4*3 = 68$ | 1 | Yes |
| <code>univ[1][5]</code> | $16+4*5 = 36$ | 0 | No |
| <code>univ[2][-1]</code> | $56+4*-1 = 52$ | 9 | No |
| <code>univ[3][-1]</code> | ?? | ?? | No |
| <code>univ[1][12]</code> | $16+4*12 = 64$ | 1 | No |
- Code does not do any bounds checking
 - Ordering of elements in different arrays not guaranteed

22

Using Nested Arrays

- Strengths**
 - C compiler handles doubly subscripted arrays
 - Generates very efficient code
 - > Avoids multiply in index computation
- Limitation**
 - Only works if have fixed array size



```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

23

Dynamic Nested Arrays

- Strength**
 - Can create matrix of arbitrary size
- Programming**
 - Must do index computation explicitly
- Performance**
 - Accessing single element costly
 - Must do multiplication

```
int * new_var_matrix(int n)
{
    return (int *)
        calloc(n*n, sizeof(int));
}
```

```
int var_ele
(int *a, int i,
int j, int n)
{
    return a[i*n+j];
}
```

```
movl 12(%ebp),%eax # i
movl 8(%ebp),%edx # a
imull 20(%ebp),%eax # n*i
addl 16(%ebp),%eax # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

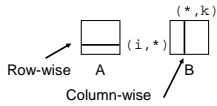
24

Dynamic Array Multiplication

- Without Optimizations

- Multiplies
 - » 2 for subscripts
 - » 1 for data
- Adds
 - » 4 for array indexing
 - » 1 for loop index
 - » 1 for data

```
/* Compute element i,k of
variable matrix product */
int var_prode_ele
(int *a, int *b,
 int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```



25

Optimizing Dynamic Array Mult.

- Optimizations

- Performed when set optimization level to -O2

- Code Motion

- Expression $i*n$ can be computed outside loop

- Strength Reduction

- Incrementing j has effect of incrementing $j*n+k$ by n

- Performance

- Compiler can optimize regular access patterns

```
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```

```
{
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++) {
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }
    return result;
}
```

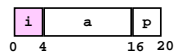
Structures

- Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

Memory Layout



- Accessing Structure Member

```
void
set_i(struct rec *r,
      int val)
{
    r->i = val;
}
```

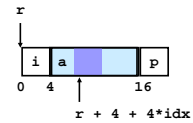
Assembly

```
# %eax = val
# %edx = r
movl %eax, (%edx) # Mem[r] = val
```

27

Generating Pointer to Struct. Member

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```



- Generating Pointer to Array Element

- Offset of each structure member determined at compile time

```
int *
find_a
(struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

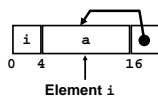
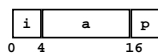
28

Structure Referencing (Cont.)

- C Code

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

```
void
set_p(struct rec *r)
{
    r->p =
    &r->a[r->i];
}
```



Element i

```
# %edx = r
movl (%edx), %ecx # r->i
leal 0(,%ecx,4), %eax # 4*(r->i)
leal 4(%edx,%eax), %eax # r+4+4*(r->i)
movl %eax, 16(%edx) # Update r->p
```

29

Alignment

- Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
 - » treated differently by Linux and Windows!

- Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
 - » Inefficient to load or store datum that spans quad word boundaries
 - » Virtual memory very tricky when datum spans 2 pages
 - Why?

- Compiler

- Inserts gaps in structure to ensure correct alignment of fields

30

Specific Cases of Alignment

- **Size of Primitive Data Type:**
 - 1 byte (e.g., char)
 - » no restrictions on address
 - 2 bytes (e.g., short)
 - » lowest 1 bit of address must be 0₂
 - 4 bytes (e.g., int, float, char *, etc.)
 - » lowest 2 bits of address must be 00₂
 - 8 bytes (e.g., double)
 - » Windows (and most other OS's & instruction sets):
 - lowest 3 bits of address must be 000₂
 - » Linux:
 - lowest 2 bits of address must be 00₂
 - i.e., treated the same as a 4-byte primitive data type
 - 12 bytes (long double)
 - » Linux:
 - lowest 2 bits of address must be 00₂
 - i.e., treated the same as a 4-byte primitive data type

31

Satisfying Alignment with Structures

- **Offsets Within Structure**
 - Must satisfy element's alignment requirement
- **Overall Structure Placement**
 - Each structure has alignment requirement K
 - » Largest alignment of any element
 - Initial address & structure length must be multiples of K
- **Example (under Windows):**
 - K = 8, due to double element

```

struct S1 {
    char c;
    int i[2];
    double v;
} *p;
  
```

32

Linux vs. Windows

Windows (including Cygwin):

- K = 8, due to double element

```

struct S1 {
    char c;
    int i[2];
    double v;
} *p;
  
```

Linux:

- K = 4; double treated like a 4-byte data type

33

Overall Alignment Requirement

```

struct S2 {
    double x;
    int i[2];
    char c;
} *p;
  
```

p must be multiple of:
8 for Windows
4 for Linux

```

struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;
  
```

p must be multiple of 4 (in either OS)

34

Ordering Elements Within Structure

```

struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
  
```

10 bytes wasted space in Windows

```

struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
  
```

2 bytes wasted space

35

Comment: Ordering

- **Can the compiler do the re-ordering**
 - NO!
 - it could IF you never did pointer offsets
 - » C allows pointer + offset style calculations
 - » hence the C compiler can't reorder
 - » hence it becomes the job of the good programmer
 - who understands what happens down at the machine level
- **Call this a blatant advert for cs4400**
 - You'll be happy you know this when you write a big program that uses most of a machine's memory...

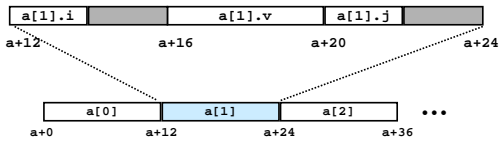
36

Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```



37

Accessing Element within Array

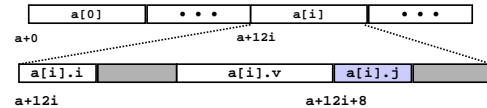
- Compute offset to start of structure
 - Compute $12 \cdot i$ as $4 \cdot (i \cdot 2)$
- Access element according to its offset within structure
 - Offset by 8
 - Assembler gives displacement as $a + 8$
 - Linker must set actual value

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```

```
short get_j(int idx)
{
    return a[idx].j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

move and sign extend word to double



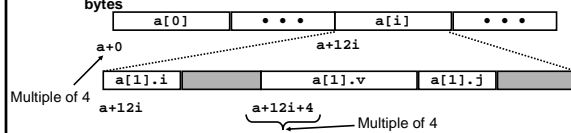
38

Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
 - a must be multiple of 4
- Offset of element within structure must be multiple of element's alignment requirement
 - v 's offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
 - Structure padded with unused space to be 12 bytes

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```



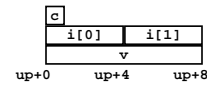
39

Union Allocation

Principles

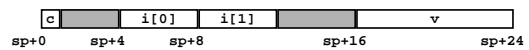
- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```



```
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

(Windows alignment)



40

Using Union to Access Bit Patterns

```
typedef union {
    float f;
    unsigned u;
} bit_float_t;
```



- Get direct access to bit representation of float
- bit2float generates float with given bit pattern
 - NOT the same as (float) u
- float2bit generates bit pattern from float
 - NOT the same as (unsigned) f

```
float bit2float(unsigned u)
{
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}
```

```
unsigned float2bit(float f)
{
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

41

Cast vs. Union

Use union {int ival; float fval} fi;

- when $fi.ival = 1$
 - `printf("%f", fi.fval)` \Rightarrow 0.000000
 - `printf("%i", (float) 1)` \Rightarrow 1.000000
- when $fi.fval = 1.0$
 - `printf("%d", fi.ival)` \Rightarrow 1065353216
 - `printf("%d", (int) 1.0)` \Rightarrow 1

Hence

- casting between number types tries to preserve value
 - e.g. the bit-level representation changes accordingly
- while unions keep the same bits
 - but the interpretations can vary

42

Summary

- **Arrays in C**
 - Contiguous allocation of memory
 - Pointer to first element
 - No bounds checking
- **Compiler Optimizations**
 - Compiler often turns array code into pointer code (`zd2int`)
 - Uses addressing modes to scale array indices
 - Lots of tricks to improve array indexing in loops
- **Structures**
 - Allocate bytes in order declared
 - Pad in middle and at end to satisfy alignment
- **Unions**
 - Overlay declarations
 - Save memory
 - Can circumvent type system

43