

Structured Data:

These slides based on some provided by the authors of our textbook:
Randal Bryant & David O'Hallaron

Topics

- Arrays
- Structs
- Unions

Basic Data Types

Integral

- Stored & operated on in general 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 & operated on in floating point registers

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

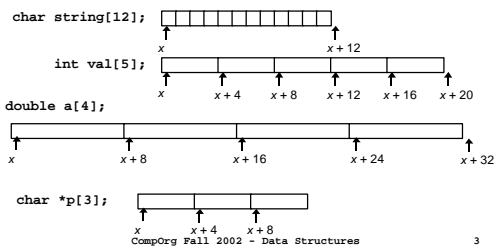
CompOrg Fall 2002 - Data Structures

2

Array Allocation

Basic Principle

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

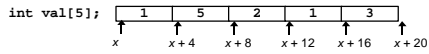


3

Array Access

Basic Principle

- T A[L];
- Array of data type T and length L
 - Identifier A can be used as a pointer to starting element of the array



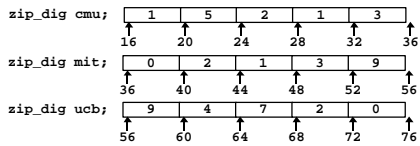
Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	x+4
&val[2]	int *	x+8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x+4i

CompOrg Fall 2002 - Data Structures

4

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 ucb = { 9, 4, 7, 2, 0 };
```



Notes

- Declaration "zip_dig cmu" equivalent to "int cmu[5]"
- Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

CompOrg Fall 2002 - Data Structures

5

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

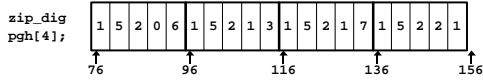
```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```

CompOrg Fall 2002 - Data Structures

6

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}};
```

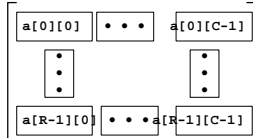


- Declaration "zip_dig pgh[4]" equivalent to "int pgh[4][5]"
 - Variable pgh denotes array of 4 elements
 - » Allocated contiguously
 - Each element is an array of 5 int's
 - » Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Nested Array Allocation

Declaration

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



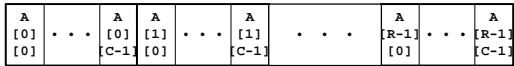
Array Size

- $R * C * K$ bytes

Arrangement

- Row-Major Ordering

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

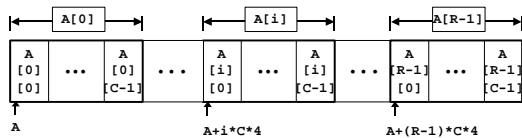


Nested Array Row Access

Row Vectors

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

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



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)
```

CompOrg Fall 2002 - Data Structures

13

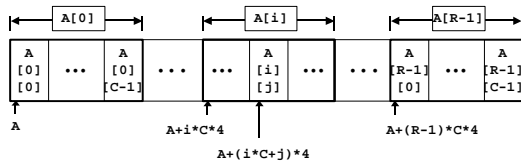
Nested Array Element Access

Array Elements

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

```
A
[i]
[j]
```

int A[R][C];



CompOrg Fall 2002 - Data Structures

14

Nested Array Element Access Code

Array Elements

- pgh[index][dig] is 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)
```

CompOrg Fall 2002 - Data Structures

15

Strange Referencing Examples

Reference	Address	Value	Guaranteed?
univ[2][3]	56+4*3 = 68	8	Yes
univ[1][5]	16+4*5 = 36	0	No
univ[2][-1]	56+4*-1 = 52	9	No
univ[3][-1]	??	??	No
univ[1][12]	16+4*12 = 64	7	No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

CompOrg Fall 2002 - Data Structures 19

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
```

CompOrg Fall 2002 - Data Structures 20

Generating Pointer to Structure 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
```

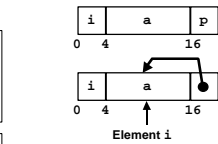
CompOrg Fall 2002 - Data Structures 21

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];
}
```



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

CompOrg Fall 2002 - Data Structures

22

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

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

CompOrg Fall 2002 - Data Structures

23

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

CompOrg Fall 2002 - Data Structures

24

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

CompOrg Fall 2002 - Data Structures 28

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];
```

CompOrg Fall 2002 - Data Structures 29

Accessing Element within Array

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

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

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

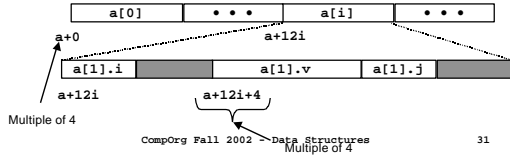
CompOrg Fall 2002 - Data Structures 30

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];
```

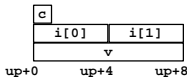


Union Allocation

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)



Using Union to Access Bit Patterns

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

```
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;
}
```

- 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

CompOrg Fall 2002 - Data Structures

33

Byte Ordering

Idea

- Long/quad words stored in memory as 4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Most significant byte has lowest address
- IBM 360/370, Motorola 68K, Sparc

Little Endian

- Least significant byte has lowest address
- Intel x86, Digital VAX

Byte Ordering Example

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]	s[1]	s[2]	s[3]				
	i[0]				i[1]		

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

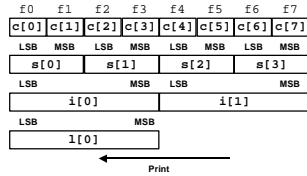
printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%x]\n",
    dw.l[0]);
```

Byte Ordering on x86

Little Endian



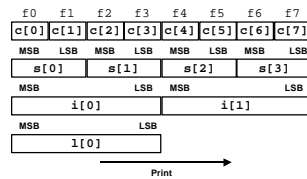
Output on Pentium:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [f3f2f1f0]

CompOrg Fall 2002 - Data Structures 37

Byte Ordering on Sun

Big Endian



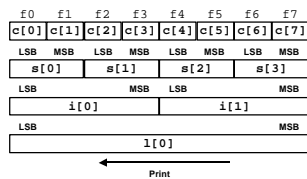
Output on Sun:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints 0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long 0 == [0xf0f1f2f3]

CompOrg Fall 2002 - Data Structures 38

Byte Ordering on Alpha

Little Endian



Output on Alpha:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

CompOrg Fall 2002 - Data Structures 39

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
- Way to circumvent type system

CompOrg Fall 2002 - Data Structures

40
