CS 105
"Tour of the Black Holes of Computing"

Machine-Level Programming IV: Structured Data

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`
quad word	`q`	8	`[unsigned] long`

Floating Point

- Stored & operated on in *floating-point* registers (not covered in CS 105)

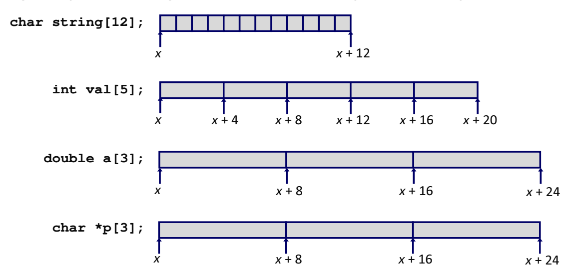
Intel	GAS	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`

Array Allocation

Basic Principle

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

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



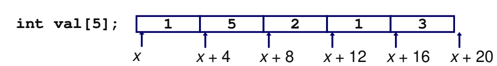
Array Access

val[i] => *(val+i)
i[val] => *(i+val)

Basic Principle

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

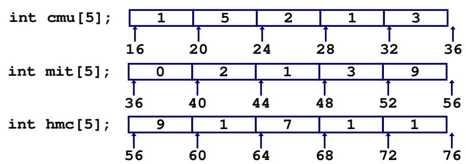
- Array of data type *T* and length *L*
- Identifier `A` can be used as a pointer to array element 0



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

Array Example

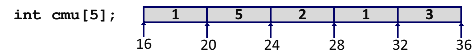
```
int cmu[5] = {1, 5, 2, 1, 3};
int mit[5] = {0, 2, 1, 3, 9};
int hmc[5] = {9, 1, 7, 1, 1};
```



Note:

- Example arrays were allocated in successive 20-byte blocks
  - Not guaranteed to happen in general
- Here, `[5]` could be written as `[]` because initializer implies size

Array Accessing Example



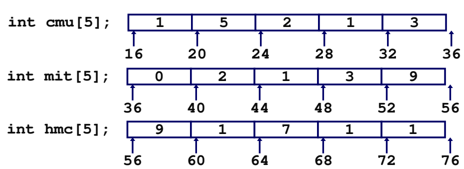

```
int get_digit(int z[], int digit)
{
  return z[digit];
}
```

x86-64

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

- As argument, size of z doesn't need to be specified
- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

Referencing Examples



Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	3	
`mit[5]`	`36 + 4* 5 = 56`	9	
`mit[-1]`	`36 + 4*-1 = 32`	3	
`cmu[15]`	`16 + 4*15 = 76`	??	

- Out-of-range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays

Array Loop Example (-O1 on an old compiler)

```
void zincr(int z[5])
{
  size_t i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax           #   i = 0
  jmp     .L3                #   goto middle
.L4:                         # loop:
  addl    $1, (%rdi,%rax,4)  #   z[i]++
  addq    $1, %rax           #   i++
.L3:                         # middle
  cmpq    $4, %rax           #   i:4
  jbe     .L4                #   if <=, goto loop
  rep; ret
```

Array Loop Example (-O1 on current gcc)

```
void zincr(int z[5])
{
  size_t i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

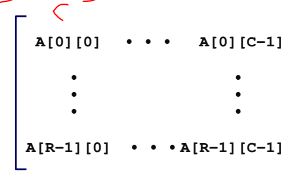
```
  # %rdi = z
  leaq    20(%rdi), %rax     #   t = &z[5]
.L2:                         # loop:
  addl    $1, (%rdi)         #   (*z)++
  addq    $4, %rdi           #   z++
  cmpq    %rax, %rdi         #   z : t
  jne     .L2                #   if !=, goto loop
  rep ret
```


Multidimensional (Nested) Arrays

Declaration

T `A[R][C];`

- 2D array of data type *T*
- *R* rows, *C* columns
- Type *T* element requires *K* bytes



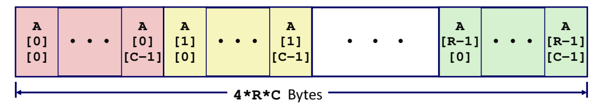
Array Size

- *R* * *C* * *K* bytes

Arrangement

- Row-Major Ordering

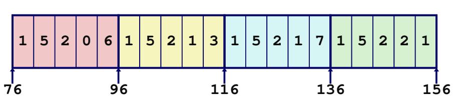
`int A[R][C];`



Nested Array Example

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

`int pgh[4][5];`



Variable `pgh`: array of 4 elements, allocated contiguously

- Each element is an array of 5 `int`'s, allocated contiguously

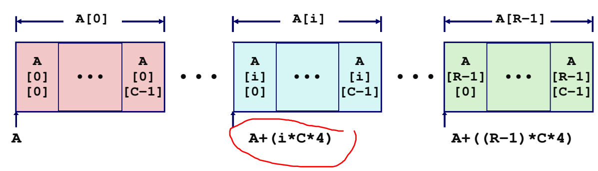
"Row-Major" ordering of all elements in memory

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];`

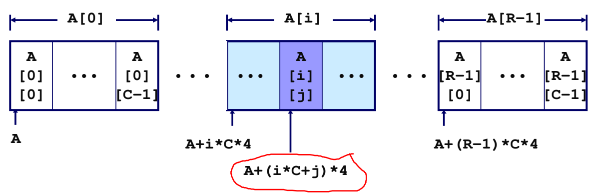


Nested Array Element Access

Array Elements

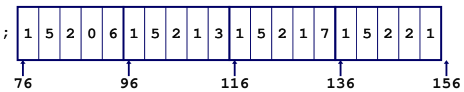
- **`A[i][j]` is element of type** *T*, which requires *K* bytes
- **Address** $A + i * (C * K) + j * K = A + (i * C + j) * K$

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



Strange Referencing Examples

```
int pgh[4][5];
```



Reference	Address	Value	Guaranteed?
`pgh[3][3]`	`76+20*3+4*3 = 148`	2	
`pgh[2][5]`	`76+20*2+4*5 = 136`	1	
`pgh[2][-1]`	`76+20*2+4*-1 = 112`	3	
`pgh[4][-1]`	`76+20*4+4*-1 = 152`	1	
`pgh[0][19]`	`76+20*0+4*19 = 152`	1	
`pgh[0][-1]`	`76+20*0+4*-1 = 72`	??	

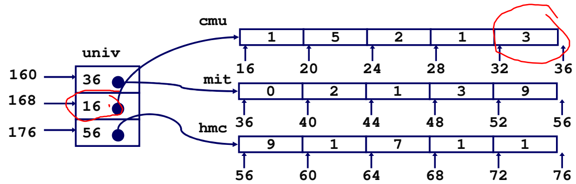
- **Code does not do any bounds checking**
- **Ordering of elements within array guaranteed**

Multi-Level Array Example

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

```
int cmu[] = {1, 5, 2, 1, 3};
int mit[] = {0, 2, 1, 3, 9};
int hmc[] = {9, 1, 7, 1, 1};
```

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



Element Access in Multi-Level Array

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

```
  salq    $2, %rsi                # 4*digit
  addq    univ(,%rdi,8), %rsi     # p = univ[index] + 4*digit
  movl    (%rsi), %eax            # return *p
  ret
```

Computation

- Element access `Mem[Mem[univ+8*index]+4*digit]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

Array Element Accesses

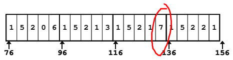
- Similar C references
- Different address computation

Nested Array

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

- Element at

`Mem[pgh+20*index+4*dig]`

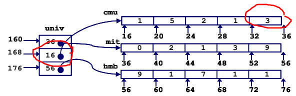


Multi-Level Array

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

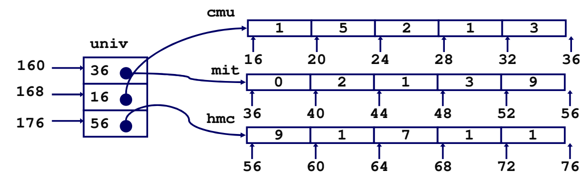
- Element at

`Mem[Mem[univ+4*index]+4*dig]`



105

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	1	
`univ[1][5]`	`16+4*5 = 36`	0	
`univ[2][-1]`	`56+4*-1 = 52`	9	
`univ[3][-1]`	`??`	??	
`univ[1][12]`	`16+4*12 = 64`	7	

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

105

N x N Matrix Code

- Fixed dimensions
  - Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
  return a[i][j];
}
```

- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
  return a[IDX(n,i,j)];
}
```

- Variable dimensions, implicit indexing
  - Now supported by gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```


105

16 X 16 Matrix Access

- Array Elements
  - Address **A** + $i * (C * K) + j * K$
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j)
{
  return a[i][j];
}
```

```
  # a in %rdi, i in %rsi, j in %rdx
  salq    $6, %rsi               # 64*i
  addq    %rsi, %rdi             # a + 64*i
  movl    (%rdi,%rdx,4), %eax    # M[a + 64*i + 4*j]
  ret
```


105

N x N Matrix Access

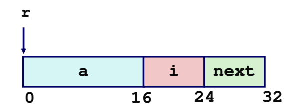
- **Array Elements**
  - Address **`A`** **+** *i* * (*C* * *K*) + *j* * *K*
  - C = n, K = 4
  - Must perform integer multiplication

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq    %rdx, %rdi           # n*i
  leaq     (%rsi,%rdi,4), %rax  # a + 4*n*i
  movl     (%rax,%rcx,4), %eax  # a + 4*n*i + 4*j
  ret
```


Structure Representation

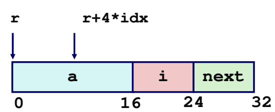
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as `r + 4*idx`

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

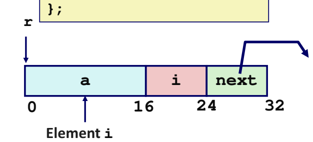
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```


Following Linked List

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```

- C Code

```
void set_val(struct rec *r, int val)
{
  while (r != NULL) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
`%rdi`	`r`
`%rsi`	`val`

```
.L11:                            # loop:
  movq    16(%rdi), %rax         #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)    #   M[r+4*i] = val
  movq    24(%rdi), %rdi         #   r = M[r+24]
  testq   %rdi, %rdi             #   Test r
  jne     .L11                   #   if !=0 goto loop
```

Alignment Principles

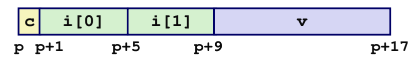
Aligned Data
- Primitive data type requires *K* bytes
- Address must be multiple of *K*
- Required on some machines; advised on x86-64

Motivation for Aligning Data
- Memory accessed by (aligned) chunks of 4 or 8 bytes (system-dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

Compiler
- Inserts gaps in structure to ensure correct alignment of fields

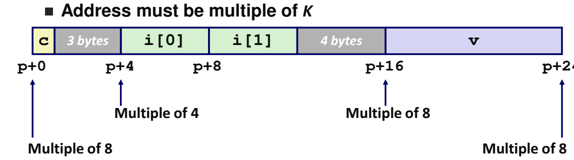
Structures & Alignment

Unaligned Data




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

Aligned Data
- Primitive data type requires *K* bytes
- Address must be multiple of *K*



Specific Cases of Alignment (x86-64)

1 byte: `char`, …
- no restrictions on address

2 bytes: `short`, …
- lowest 1 bit of address must be 0_2

4 bytes: `int`, `float`, …
- lowest 2 bits of address must be 00_2

8 bytes: `double`, `long`, `char *`, …
- lowest 3 bits of address must be 000_2

16 bytes: `long double` (GCC on Linux)
- lowest 4 bits of address must be 0000_2

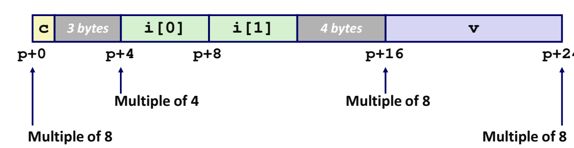
Satisfying Alignment Within Structures

Within structure:
- Must satisfy each element's alignment requirement

Overall structure placement
- Each structure has alignment requirement K
  - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

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

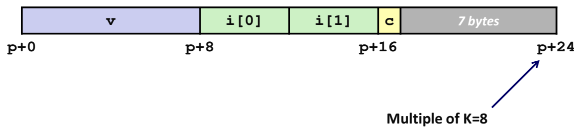
Example:
- K = 8, due to `double` element



Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

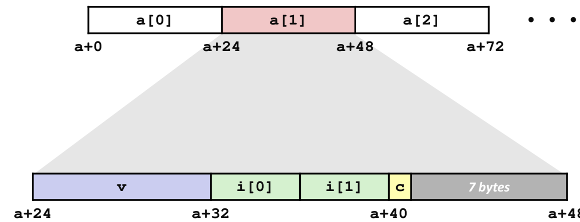
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

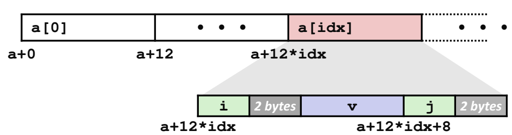
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- Compute array offset 12*idx   a+12*idx+8
  - sizeof(struct S3), including alignment spacers
- Element j is at offset 8 within structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




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

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

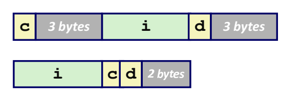

Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)

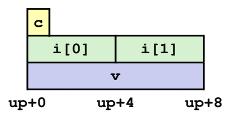


Union Allocation

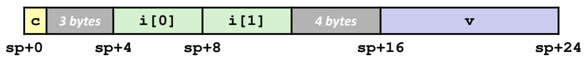
Allocate according to largest element

Can only use one field at a time

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

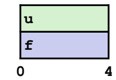



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

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




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

Same as `(float) u` ?

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

Same as `(unsigned) f` ?

Byte Ordering Revisited

Idea

- **Short/long/quad words (x86 terminology; C:** short/int/long) **stored in memory as 2/4/8 consecutive bytes**
- **Which byte is most (least) significant?**
- **Can cause problems when exchanging binary data between machines**

Big Endian

- **Most significant byte has lowest address**
- **MIPS; Internet**

Little Endian

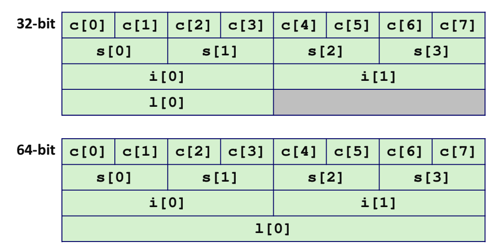
- **Least significant byte has lowest address**
- **Intel x86, ARM Android and IOS**

Bi Endian

- **Can be configured either way**
- **ARM**

Byte Ordering Example

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



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%lx]\n",
    dw.l[0]);
```


Byte Ordering on Sun

Big Endian

f0	f1	f2	f3	f4	f5	f6	f7
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]			
l[0]							

MSB LSB MSB LSB

Print →

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]
```


Byte Ordering on x86-64, ARM, MIPS

Little Endian

f0	f1	f2	f3	f4	f5	f6	f7
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]			
l[0]							

LSB MSB

← Print

Output on x86-64:

```
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]
```


Summary of Compound Types in C

Arrays
- Contiguous allocation of memory
- Aligned to satisfy every element's alignment requirement
- Pointer to first element
- No bounds checking

Structures
- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions
- Overlay declarations
- Designed to support polymorphic structures
- Way to circumvent type system