Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>Type</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Goldberg</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>d</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>

Array Allocation

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

```
char string[12];
```
```
int val[5];
```
```
double a[4];
```
```
char *p[3];
```

Array Access

Basic Principle
- Array of data type $T$ and length $L$
- Identifier $A$ can be used as a pointer to starting element of the array

```
int val[5];
```
```
char string[12];
```
```
int val[5];
```
```
double a[4];
```
```
char *p[3];
```

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 * | $x + 4i$ |
Array Example

```c
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

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)

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

Memory Reference Code

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

Array Loop Example

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

Referencing Examples

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

Code Does Not Do Any Bounds Checking!

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4* 3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4* 5 = 56</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

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

Registers
%ecx z
%eax zi
%ebx zend

Computations
• \(10 \times zi + *z\) implemented as
  \(*z + 2 \times (zi+4 \times zi)\)
• \(z++\) increments by 4

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

Nested Array Example

```c
#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}};
```

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
\( R \times C \times K \) bytes

Arrangement
• Row-Major Ordering

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

Nested Array Row Access

Row Vectors
• \( A[1] \) is array of \( C \) elements
• Each element of type \( T \)
• Starting address \( A + i \times C \times K \)

```c
int A[R][C];
```
Nested Array Row Access Code

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

Row Vector
- `pgh[index]` is an array of 5 int’s.
- Starting address `pgh + 20*index`

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

```c
int get_pgh_row(int index) {
    return pgh[index];
}
```

# Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pgh[3][3]</code></td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2 Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1 Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3 Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1 Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1 Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>?? No</td>
</tr>
</tbody>
</table>

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

Nested Array Element Access Code

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

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

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

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

Strange Referencing Examples

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

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

### Accessing Element in Multi-Level Array

**Computation**

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

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

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td>56+4*3 = 68</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td>16+4*5 = 36</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td>56+4*-1 = 52</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td>16+4*12 = 64</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

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

Using Nested Arrays

- C compiler handles doubly subscripted arrays
- Generates very efficient code
  - Avoids multiply in index computation

```c
#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;
}
```

Strengths

- C compiler handles doubly subscripted arrays
- Generates very efficient code
  - Avoids multiply in index computation

Limitation

- Only works if have fixed array size
Dynamic Nested Arrays

**Strength**
- Can create matrix of arbitrary size

**Programming**
- Must do index computation explicitly

**Performance**
- Accessing single element costly
- Must do multiplication

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

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

```assembly
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)]
```

**Dynamic Array Multiplication**

**Without Optimizations**
- Multiplies
  - 2 for subscripts
  - 1 for data
- Adds
  - 4 for array indexing
  - 1 for loop index
  - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_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;
}
```

**Optimizing Dynamic Array Multiplication**

**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

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

```c
{ 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

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

**Memory Layout**

```
0 4 16 20
```

**Accessing Structure Member**

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

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

Structure Referencing (Cont.)

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

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

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

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

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 0
- 4 bytes (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00
- 8 bytes (e.g., double)
  - Windows (and most other OS's & instruction sets):
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type
- 12 bytes (long double)
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type
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

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
</tr>
</tbody>
</table>

Multiple of 4 Multiple of 8

Multiple of 8 Multiple of 8

Effect of Overall Alignment Requirement

<table>
<thead>
<tr>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>double x;</td>
</tr>
<tr>
<td>int i[2];</td>
</tr>
<tr>
<td>char c;</td>
</tr>
<tr>
<td>} *p;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x[2];</td>
</tr>
<tr>
<td>int i[2];</td>
</tr>
<tr>
<td>char c;</td>
</tr>
<tr>
<td>} *p;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>char c1;</td>
</tr>
<tr>
<td>double v;</td>
</tr>
<tr>
<td>char c2;</td>
</tr>
<tr>
<td>int i;</td>
</tr>
<tr>
<td>} *p;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S5</th>
</tr>
</thead>
<tbody>
<tr>
<td>double v;</td>
</tr>
<tr>
<td>char c1;</td>
</tr>
<tr>
<td>char c2;</td>
</tr>
<tr>
<td>int i;</td>
</tr>
<tr>
<td>} *p;</td>
</tr>
</tbody>
</table>

Linux vs. Windows

Windows (including Cygwin):
- $K = 8$, due to double element

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
</tr>
</tbody>
</table>

Multiple of 4 Multiple of 8

Multiple of 4 Multiple of 8

Multiple of 8

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

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
</tr>
</tbody>
</table>

Multiple of 4 Multiple of 4

Multiple of 4 Multiple of 4

Multiple of 4

Ordering Elements Within Structure

<table>
<thead>
<tr>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>char c1;</td>
</tr>
<tr>
<td>double v;</td>
</tr>
<tr>
<td>char c2;</td>
</tr>
<tr>
<td>int i;</td>
</tr>
<tr>
<td>} *p;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S5</th>
</tr>
</thead>
<tbody>
<tr>
<td>double v;</td>
</tr>
<tr>
<td>char c1;</td>
</tr>
<tr>
<td>char c2;</td>
</tr>
<tr>
<td>int i;</td>
</tr>
<tr>
<td>} *p;</td>
</tr>
</tbody>
</table>

10 bytes wasted space in Windows

2 bytes wasted space
**Arrays of Structures**

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

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

**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

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

**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

```c
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

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

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

### Using Union to Access Bit Patterns

- 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`

### 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

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

```c
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 x86

**Little Endian**

|x| y| z| w| v| u| t| s| r| q| p| o| n| m| l| k| j| i| h| g| f| e| d| c| b| a|
| f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | s0 | s1 | s2 | s3 | i0 | i1 | l0 | l1 | l2 | l3 | l4 | l5 | l6 | l7 |

**Output on Pentium:**

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [f3f2f1f0]

### Byte Ordering on Sun

**Big Endian**

|x| y| z| w| v| u| t| s| r| q| p| o| n| m| l| k| j| i| h| g| f| e| d| c| b| a|
| f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | s0 | s1 | s2 | s3 | i0 | i1 | l0 | l1 | l2 | l3 | l4 | l5 | l6 | l7 |

**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 Alpha

**Little Endian**

|x| y| z| w| v| u| t| s| r| q| p| o| n| m| l| k| j| i| h| g| f| e| d| c| b| a|
| f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7 | s0 | s1 | s2 | s3 | i0 | i1 | l0 | l1 | l2 | l3 | l4 | l5 | l6 | l7 |

**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]

### 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
- 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