Basic Data Types

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

<table>
<thead>
<tr>
<th></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>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>GAS</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>l</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>

Array Access

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

```
char string[12];  // At x + 0
int val[5];      // At x + 0
double a[3];     // At x + 0
char *p[3];      // At x + 0
```

**Reference**
- Identifier $A$ can be used as a pointer to array element 0

```
reference_type_value = *(val + i);
```

Array Allocation

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

```
char string[12];  // At x + 0
int val[5];      // At x + 0
double a[3];     // At x + 0
char *p[3];      // At x + 0
```
**Array Example**

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

```c
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

```c
int cmu[5];
int mit[5];
int hmc[5];
```

**Referencing Examples**

Code Does Not Do Any Bounds Checking!

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4*3 = 48 3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4*5 = 56 9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32 3</td>
<td>Yes</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76 ??</td>
<td>No</td>
</tr>
</tbody>
</table>
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

**Array Accessing Example**

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

**Array Loop Example (-O1 on an old compiler)**

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

```c
void zincr(int z[5]) {
    size_t i;
    for (i = 0; i < 5; i++)
        z[i]++;
}
```
Array Loop Example (-O1 on current gcc)

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

Multidimensional (Nested) Arrays

**Declaration**
- 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

```
#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, 1, 1}};
```

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, 1, 1}};
```

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];
```

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 Element Access

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

Strange Referencing Examples

Reference Address Value Guaranteed?

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg</td>
<td>h[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
</tr>
<tr>
<td>pg</td>
<td>h[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
</tr>
<tr>
<td>pg</td>
<td>h[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
</tr>
<tr>
<td>pg</td>
<td>h[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pg</td>
<td>h[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pg</td>
<td>h[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
</tr>
</tbody>
</table>

- 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

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 *pret
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

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

Element at Mem[pgh+20*index+4*dig]

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

Element at Mem[Mem[univ+4*index]+4*dig]

Strange Referencing Examples

- Nested Array
- Multi-Level Array

```c
#define IDX(n, i, j) ((i)*(n)+(j))
```

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

```c
#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];
}
```

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

16 X 16 Matrix Access

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

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

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

```c
#define 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 # N[a + 64*i + 4*j]
ret
```

N x N Matrix Code

- Fixed dimensions
  - Know value of N at compile time
- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays
- Variable dimensions, implicit indexing
  - Now supported by gcc

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
### N x N Matrix Access

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

```c
/* 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];
}
```

### Structure Representation

- 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

- Offset of each structure member determined at compile time
- Compute as $r + 4*\text{idx}$

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

### Following Linked List

#### C Code

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

#### Diagram

- ![Diagram of a linked list](diagram.png)

### Generating Pointer to Array Element

- Offset of each array element determined at compile time
- Compute as $r + 4*\text{idx}$
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

Specific Cases of Alignment (x86-64)
1 byte: char, ...
  - No restrictions on address
2 bytes: short, ...
  - Lowest 1 bit of address must be 0
4 bytes: int, float, ...
  - Lowest 2 bits of address must be 00
8 bytes: double, long, char *, ...
  - Lowest 3 bits of address must be 000
16 bytes: long double (GCC on Linux)
  - Lowest 4 bits of address must be 0000

Structures & Alignment

Unaligned Data

Aligned Data

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

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 $k$

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

Meeting Overall Alignment Requirement

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

Arrays of Structures

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

Accessing Array Elements

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

Saving Space

Put large data types first

Effect ($K=4$)
Union Allocation

Allocate according to largest element
Can only use one field at a time

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

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

Using Union to Access Bit Patterns

```c
typedef union {
    float f;
    unsigned int 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;
}
```

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 (Cont.)

```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 Sun

Big Endian

```
<table>
<thead>
<tr>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
<td>f4</td>
<td>f5</td>
<td>f6</td>
<td>f7</td>
</tr>
</tbody>
</table>
```

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** == [0xf0f1f2f3f4f5f6f7]

Byte Ordering on x86-64, ARM, MIPS

Little Endian

```
<table>
<thead>
<tr>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
<td>f4</td>
<td>f5</td>
<td>f6</td>
<td>f7</td>
</tr>
</tbody>
</table>
```

Output on x86-64:
- **Characters 0-7** == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts 0-3** == [0xf0f1f2f3, 0xf4f5f6, 0xf7f0f6]
- **Ints 0-1** == [0xf3f2f1f0, 0xf6f5f4f3f2f1f0]
- **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