Assembly Programming: Data
Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structures**
  - Allocation
  - Access
  - Alignment
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

```plaintext
char string[12];

int val[5];

double a[3];

char *p[3];
```
Array Access

- **Basic Principle**
  
  ```c
  T A[L];
  ```

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

- **Reference**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>x + 4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + 4 i</td>
</tr>
</tbody>
</table>

---

**Diagram:**

```
int val[5];
```

```
x x + 4 x + 8 x + 12 x + 16 x + 20
```
Array Example

```c
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };```

- 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

- Register %rdi contains starting address of array
- Register %rsi contains array index
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi,%rsi,4)

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

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

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

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

- **Declaration**
  - $T \mathbf{A}[R][C]$;
  - 2D 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

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

- Array Size:
  - $R \times C \times K$ bytes

- Arrangement:
  - Row-Major Ordering

```
A[0][0] \ldots A[0][C-1]
  \vdots
A[R-1][0] \ldots A[R-1][C-1]
```

```
4 * R * C Bytes
```
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 }};
```

- “zip_dig pgh[4]” equivalent to “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 $A + i \times (C \times K)$

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

![Diagram showing row access in a 2D array]
**Nested Array Row Access Code**

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

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

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

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax  # 5 * index
leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```
Nested Array Element Access

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

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

![Diagram of nested array element access](image)
Nested Array Element Access Code

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

```
leaq (%rdi,%rdi,4), %rax  # 5*index
addl %rax, %rsi           # 5*index+dig
movl pgh(,%rsi,4), %eax  # M[pgh + 4*(5*index+dig)]
```

- **Array Elements**
  - `pgh[index][dig]` is int
  - Address: `pgh + 20*index + 4*dig`
    - `* = pgh + 4*(5*index + dig)`
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

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };,
zip_dig mit = { 0, 2, 1, 3, 9 },
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```
Element Access in Multi-Level Array

```c
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

Nested array

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

Multi-level array

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

Accesses looks similar in C, but address computations very different:

```
Mem[pgh+20*index+4*digit]   Mem[Mem[univ+8*index]+4*digit]
```
Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structures**
  - Allocation
  - Access
  - Alignment
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 a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```
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 \( r + 4 \times \text{idx} \)

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

Following Linked List

C Code

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

Element i

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
<tr>
<td>%rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

.L11:  # loop:
    movslq 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
Structures & Alignment

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

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

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

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

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

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

![Diagram of memory layout](image)

- `v` at `p+0`
- `i[0]` at `p+8`
- `i[1]` at `p+16`
- `c` at `p+24`

7 bytes

Multiple of K=8
Arrays of Structures

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

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

- Put large data types first

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

- Effect (K=4)

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

- 3 bytes
- 2 bytes