# Machine-Level Programming IV: Data Structures

Kai Shen

1

## Basic Data Types

- Integer
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on software interpretation and manipulation

| Intel | ASM | Bytes | C |
|---|---|---|---|
| byte | `b` | 1 | `[unsigned] char` |
| word | `w` | 2 | `[unsigned] short` |
| double word | `l` | 4 | `[unsigned] int` |
| quad word | `q` | 8 | `[unsigned] long int` (x86-64) |

- Floating Point
  - Stored & operated on in floating point registers

| Intel | ASM | Bytes | C |
|---|---|---|---|
| Single | `s` | 4 | `float` |
| Double | `l` | 8 | `double` |
| Extended | `t` | 10/12/16 | `long double` |

2

## Outline: Data Structures

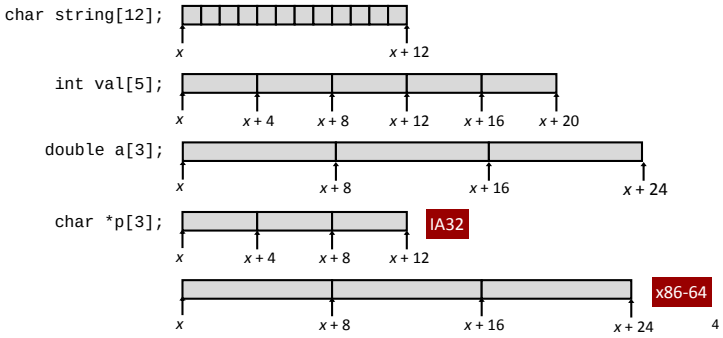
- Basic data types
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Optimizations
- Structures

3

## Array Allocation

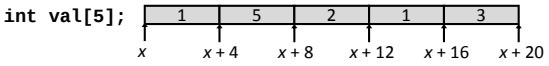
*T* **`A[`***L***`];`**

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




4

## Array Access




- Reference / Type / Value

| 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* + 4 *i* |

5

## Array Accessing Example


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

IA32

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

- Register **%edx** contains starting address of array
- Register **%eax** contains array index
- Desired value at **4*%eax + %edx**
- Use memory reference **(%edx,%eax,4)**

6

## Array Loop Example (IA32)

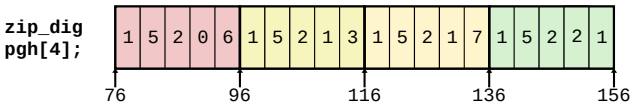
```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # edx = z
   movl  $0, %eax           #   %eax = i
.L4:                        # loop:
   addl  $1, (%edx,%eax,4)  #   z[i]++
   addl  $1, %eax           #   i++
   cmpl  $5, %eax           #   i:5
   jne   .L4                #   if !=, goto loop
```

7

## Nested Array Data Layout

```
#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 guaranteed

8

## Nested Array Row Access Code

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

```
#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 }};
```

- Row vector
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- Machine code
  - Computes and returns address
  - Using **`lea`**, compute as **`pgh + 4*(index+4*index)`**

```
# %eax = index
leal (%eax,%eax,4),%eax  # 5 * index
leal pgh(,%eax,4),%eax   # pgh + (20 * index)
```

9

## Nested Array Element Access Code

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

- Array elements
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
- Machine code
  - Computes address **`pgh + 4*((index+4*index)+dig)`**

```
movl  8(%ebp), %eax          # index
leal  (%eax,%eax,4), %eax    # 5*index
addl  12(%ebp), %eax         # 5*index+dig
movl  pgh(,%eax,4), %eax     # offset 4*(5*index+dig)
```

10

## 16 X 16 Matrix Access

```
#define N 16
typedef int fix_matrix[N][N];
```

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

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

```
movl  12(%ebp), %edx    # i
sall  $6, %edx          # i*64
movl  16(%ebp), %eax    # j
sall  $2, %eax          # j*4
addl  8(%ebp), %eax     # a + j*4
movl  (%eax,%edx), %eax # *(a + j*4 + i*64)
```

11

## n X n Matrix Access

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

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

```
movl  8(%ebp), %eax     # n
sall  $2, %eax          # n*4
movl  %eax, %edx        # n*4
imull 16(%ebp), %edx    # i*n*4
movl  20(%ebp), %eax    # j
sall  $2, %eax          # j*4
addl  12(%ebp), %eax    # a + j*4
movl  (%eax,%edx), %eax # *(a + j*4 + i*n*4)
```

12

## Outline: Data Structures

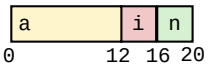
- Basic data types
- Arrays
- Structures
  - Allocation
  - Access

13

## Structure Allocation

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

Memory Layout

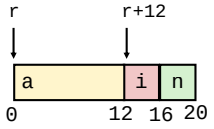



- Concept
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

14

## Structure Access

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




- Accessing Structure Member
  - Pointer indicates first byte of structure
  - Access elements with offsets

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

IA32 Assembly

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```

15

## Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

16