Lecture 15

Data Structures

CPSC 275
Introduction to Computer Systems

Basic Data Types

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

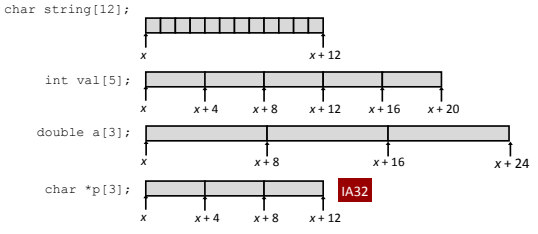
Intel	ASM	Bytes	C
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`

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

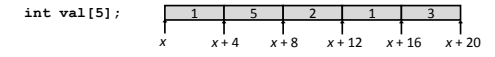
Intel	ASM	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`

Array Allocation

- Basic Principle
  T **`A[`***L***`];`**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes

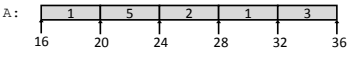


Array Access



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*

Array Accessing Example




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

IA32:

```
movl (%edx,%ecx,4),%eax
```

- Suppose that register `%edx` contains starting address of array
- Register `%ecx` contains array index
- Desired digit at `4*%ecx + %edx`
- Use memory reference `(%edx,%ecx,4)`

Array Loop Example

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

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

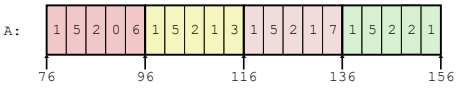
Pointer Loop Example

```
void zincr_p(int z[]) {
  int *zend = z + ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

```
                          # edx = z = p  (char pointer)
  movl  $0, %eax          #   i = 0
.L8:                      # loop:
  incl  (%edx,%eax)       #   increment *(p+i)
  addl  $4, %eax          #   i += 4
  cmpl  $20, %eax         #   compare
  jne   .L8               #   if !=, loop
```

Multi-Dimensional Arrays

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



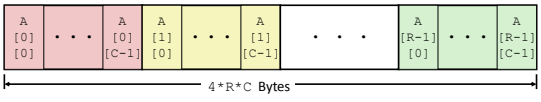
- Variable **A**: array of 4 elements, allocated contiguously
- Each element is an array of 5 **int**'s, allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Multi-Dimensional Arrays, cont'd

- 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

```
A[0][0]   . . .   A[0][C-1]
   .                  .
   .                  .
   .                  .
A[R-1][0] . . . A[R-1][C-1]
```

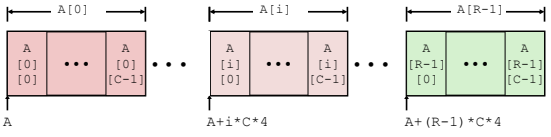
int A[R][C];



Multi-Dimensional 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];



Row Access Code

```
int *get_row(int row)
{
  return A[row];
}
```

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

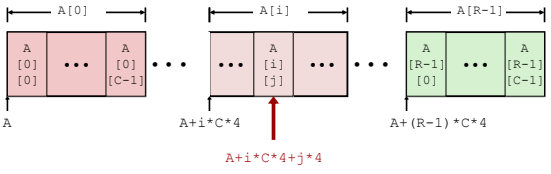
```
                          # %eax = row
leal (%eax,%eax,4),%eax   # 5 * row
leal A(,%eax,4),%eax      # A + (20 * row)
```

- Row Vector
  - **A[row]** is an array of 5 **int**'s
  - Starting address **A + 20*row**

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];



Nested Array Element Access Code

```
int get_element (int row, int col)
{
  return A[row][col];
}
```

```
movl    8(%ebp), %eax          # row
leal    (%eax,%eax,4), %eax    # 5*row
addl    12(%ebp), %eax         # 5*row+col
movl    A(,%eax,4), %eax       # offset 4*(5*row+col)
```

- Array Elements

  A[row][col] is **int**

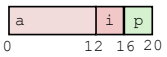
  – Address: **A + 20*row + 4*col**

  = A + 4*(5*row + col)

Structure Allocation

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

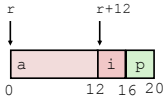
Memory Layout



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

Structure Access

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



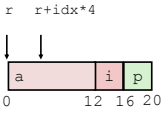
- Accessing Structure Member
  - Pointer indicates first byte of structure
  - Access elements with offsets

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

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

Generating Pointer to Structure Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Arguments
    - Mem[%ebp+8]: **r**
    - Mem[%ebp+12]: **idx**

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

```
movl    12(%ebp),%eax    # Get idx
sall    $2,%eax          # idx*4
addl    8(%ebp),%eax     # r+idx*4
```

Practice Problems

- Read CSaPP Sec. 3.8.1-3.8.4, 3.9.1 and try the following problems:

  3.35, 3.36, 3.37