Machine-Level Programming IV: Data

CS-281: Introduction to Computer Systems

Instructor:
Thomas C. Bressoud

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`
quad word	`q`	8	`[unsigned] long int` (x86-64)

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

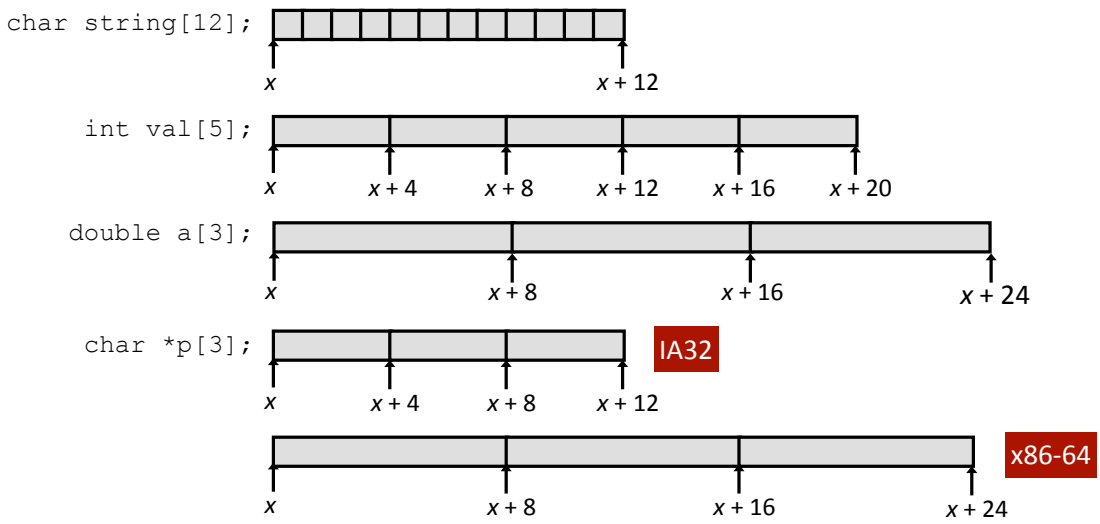
Intel	ASM	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

Array Allocation

- Basic Principle

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

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

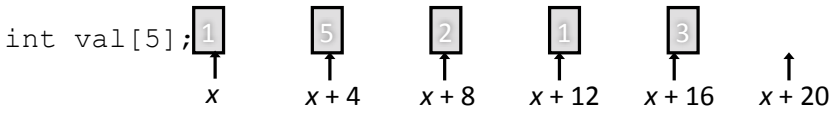


Array Access

- Basic Principle

  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**



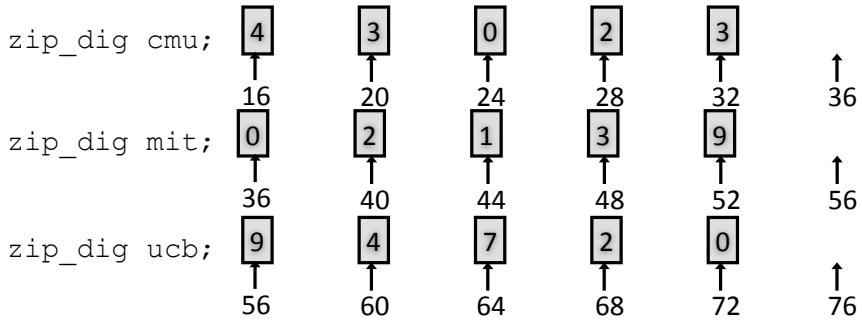
- Reference

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 Example

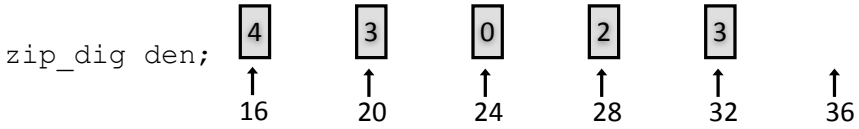
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig den = { 4, 3, 0, 2, 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




```
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 digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

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
```


Pointer Loop Example (IA32)

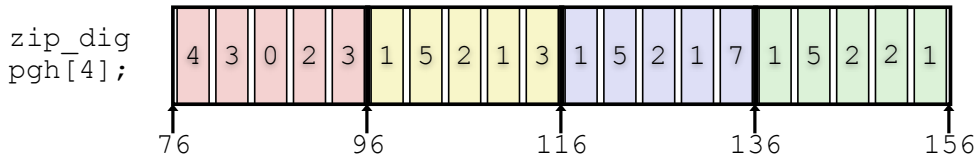
```
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

```
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
}
```

```
  # edx = z = vz
  movl  $0, %eax            #   i = 0
.L8:                        # loop:
  addl  $1, (%edx,%eax)     #   Increment vz+i
  addl  $4, %eax            #   i +=  4
  cmpl  $20, %eax           #   Compare i:20
  jne   .L8                 #   if !=, goto loop
```

Nested Array Example

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

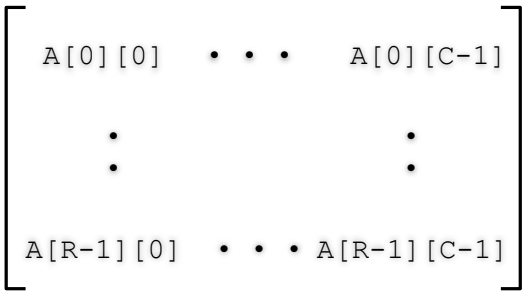


- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements guaranteed

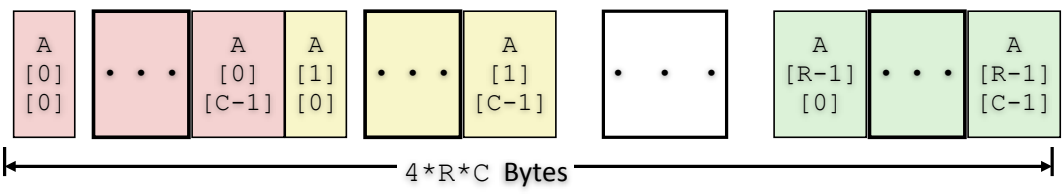
Multidimensional (Nested) Arrays

- 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



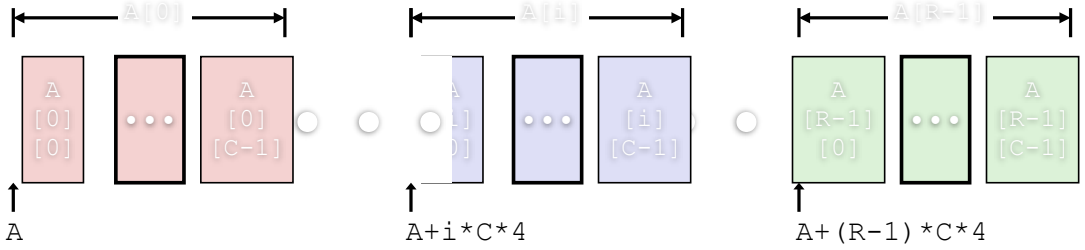
`int A[R][C];`



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];
```



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 }};
```

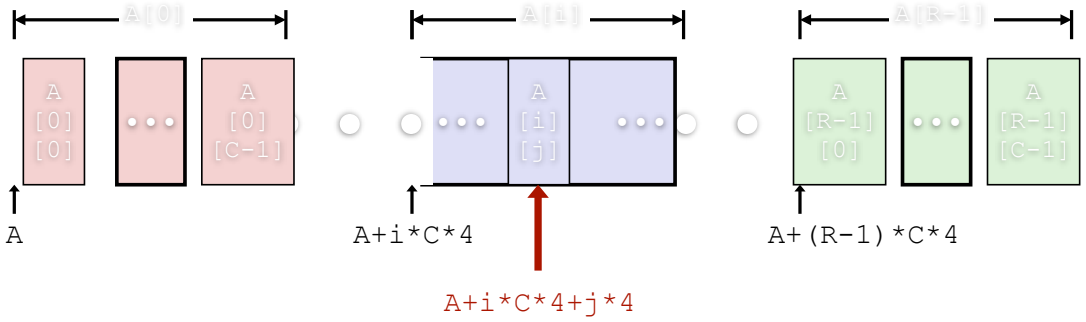
```
                # %eax = index
        leal (%eax,%eax,4),%eax # 5 * index
  leal pgh(,%eax,4),%eax # pgh + (20 * index)
```

- Row Vector
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- IA32 Code
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

Nested Array Row 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_pgh_digit
  (int index, int dig)
{
  return pgh[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)
```

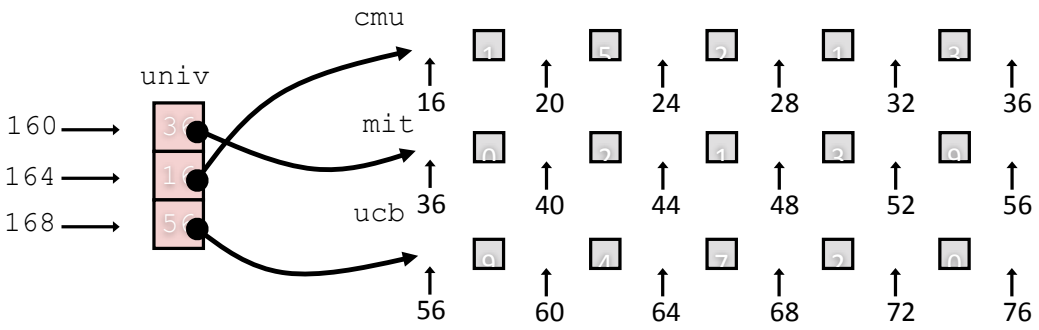
- Array Elements
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**
- IA32 Code
  - Computes address **`pgh + 4*((index+4*index)+dig)`**

Multi-Level Array Example

```
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};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s



Element Access in Multi-Level Array

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

```
    movl 8(%ebp), %eax   # index
    movl univ(,%eax,4), %edx # p = univ[index]
    movl 12(%ebp), %eax   # dig
    movl (%edx,%eax,4), %eax # p[dig]
```

- Computation (IA32)
  - Element access **`Mem[Mem[univ+4*index]+4*dig]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

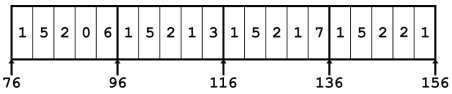
Array Element Accesses

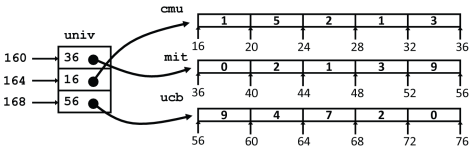
Nested array

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

Multi-level array

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





Accesses looks similar in C, but addresses very different:

`Mem[pgh+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

N X N Matrix Code

- Fixed dimensions
  - Know value of N at compile time

```
#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];
}
```

- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays

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

- Variable dimensions, implicit indexing
  - Now supported by gcc

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

16 X 16 Matrix Access

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

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

```
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)
```

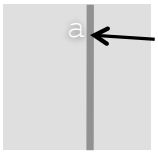

n X n Matrix Access

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

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

```
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)
```

Optimizing Fixed Array Access




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

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

- Computation
  - Step through all elements in column j
- Optimization
  - Retrieving successive elements from single column

Optimizing Fixed Array Access

- Optimization
  - Compute ajp = &a[i][j]
    - Initially = a + 4*j
    - Increment by 4*N

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

```
.L8:   # loop:
    movl (%ecx), %eax #   Read *ajp
movl %eax, (%ebx,%edx,4) #   Save in dest[i]
    addl $1, %edx #   i++
  addl $64, %ecx #   ajp += 4*N
    cmpl $16, %edx #   i:N
  jne .L8 #   if !=, goto loop
```

Optimizing Variable Array Access

- Compute ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*n

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = 0; i < n; i++)
    dest[i] = a[i][j];
}
```

```
.L18:  # loop:
    movl (%ecx), %eax #   Read *ajp
    movl %eax, (%edi,%edx,4) #   Save in dest[i]
    addl $1, %edx #   i++
    addl $ebx, %ecx #   ajp += 4*n
    cmpl $edx, %esi #   n:i
    jg .L18 #   if >, goto loop
```

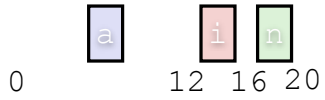

Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access

Structure Allocation

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

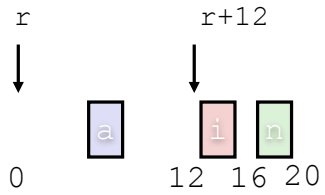
Memory Layout



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

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

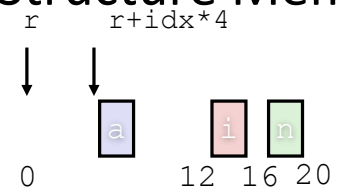
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

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

Generating Pointer to Structure Member

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



- 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];
}
```

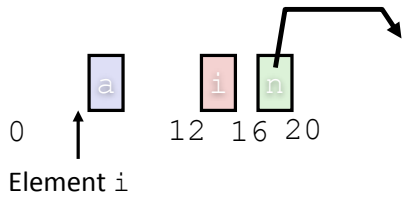
```
movl 12(%ebp), %eax # Get idx
sall $2, %eax # idx*4
addl 8(%ebp), %eax # r+idx*4
```


Following Linked List

- C Code

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

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



Register	Value
%edx	r
%ecx	val

```
.L17:  # loop:
  movl 12(%edx), %eax # r->i
  movl %ecx, (%edx,%eax,4) # r->a[i] = val
  movl 16(%edx), %edx # r = r->n
  testl %edx, %edx # Test r
  jne .L17 # If != 0 goto loop
```

Summary

- Procedures in x86-64
  - Stack frame is relative to stack pointer
  - Parameters passed in registers
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access