CS429: Computer Organization and Architecture

Instruction Set Architecture V

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: September 23, 2019 at 12:37

Basic Data Types

Integral

- Stored and operated on in general registers.
- Signed vs. unsigned depends on instructions used.

Intel	GAS	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

Floating Point

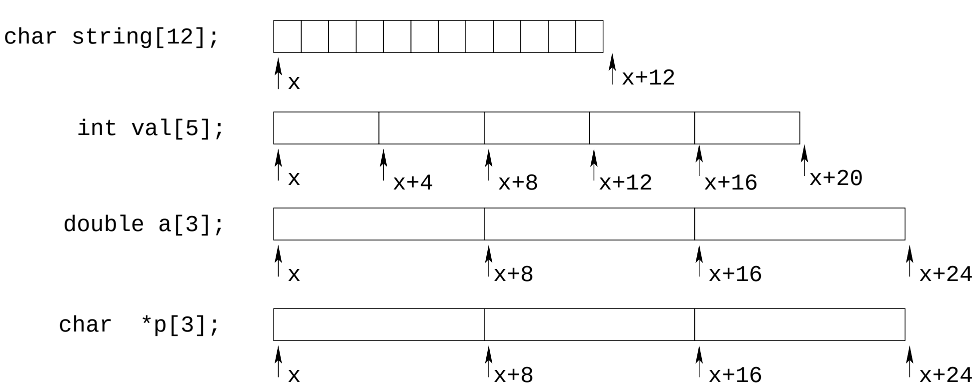
Stored and operated on in floating point registers.

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

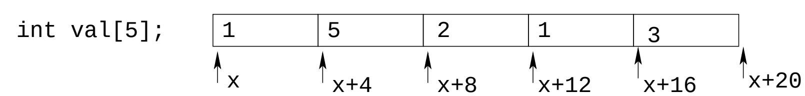
Array Allocation

Basic Principle: `T A[L]`

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



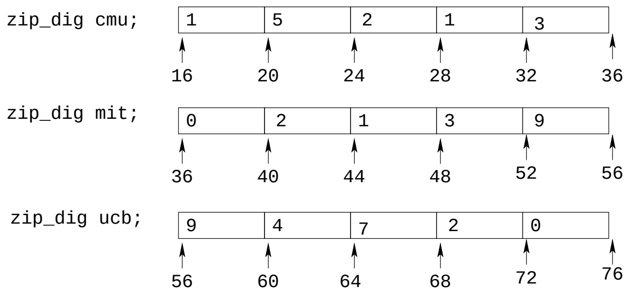
Array Access



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+j`	`int *`	$x + 4j$

Note the use of pointer arithmetic.

Array Example




```
#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` is equivalent to `int cmu[5]`.

Example arrays were allocated in successive 20 byte block.

That's not guaranteed to happen in general.

Array Accessing Example

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

Memory Reference Code

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

Computation

- Register `%rdi` contains the starting address of the array.
- Register `%rsi` contains the array index.
- The desired digit is at `%rdi + (4 * %rsi)`.
- User memory reference `(%rdi,%rsi,4)`.

Array Loop Example

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

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


Multidimensional (Nested) Arrays

Declaration: `T A[R][C];`

- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A][R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

Array Size: $R * C * K$ bytes

Arrangement: Row-Major ordering (guaranteed)

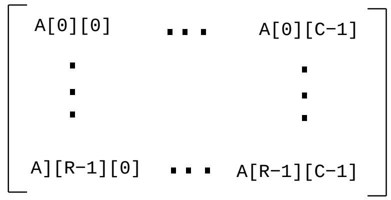
Row major order means the elements are stored in the following order:

$$[A_{0,0}, \ldots, A_{0,C-1}, A_{1,0}, \ldots, A_{1,C-1}, \ldots, A_{R-1,0}, \ldots, A_{R-1,C-1}].$$

Multidimensional Array Access

Declaration: `T A[R][C];`

- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

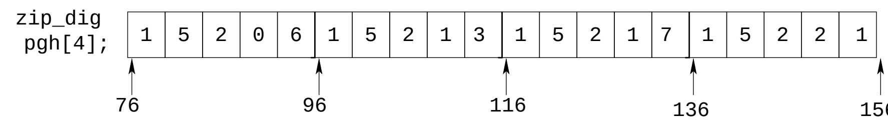


To access element A[i][j], perform the following computation:

$$A + i * C * K + j * K$$

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}};
```



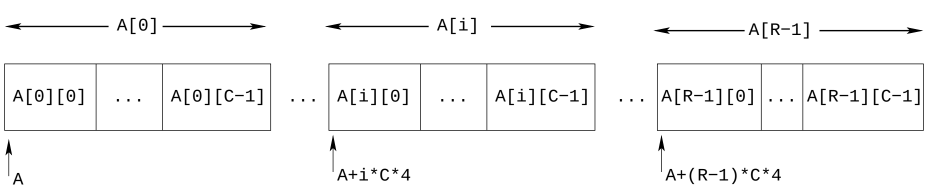
- Declaration "`zip_dig pgh[4]`" is equivalent to "`int pgh[4][5]`."
- Variable `pgh` denotes an array of 4 elements allocated contiguously.
- Each element is an array of 5 ints, which are allocated contiguously.
- This is "row-major" ordering of all elements, guaranteed.

Nested Array Row Access

Row Vectors:

Given a nested array declaration `T A[R][C]`, you can think of this as an array of arrays.

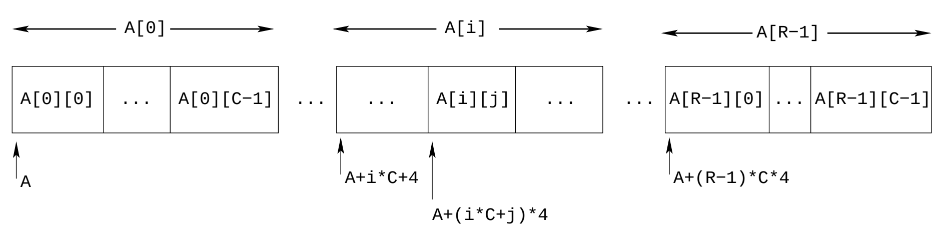
- `A[i]` is an array of C elements.
- Each element of `A[i]` has type T, and requires K bytes.
- The starting address of `A[i]` is `A + i * C * K`.



Nested Array Element Access

Array Elements

- `A[i][j]` is an element of type T, which requires K bytes.
- The address is `A + (i * C + j)*K`.

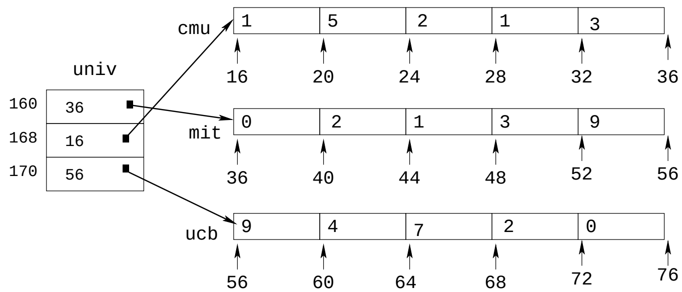


Multi-Level Array Example

```
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 an array of 3 elements.
- Each element is a pointer (8 bytes).
- Each pointer points to an array of ints (may vary in length; i.e., ragged array is possible).



Element Access in a Multi-Level Array

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

Computation

- Element access Mem[Mem[univ+8*index] + 4*dig]
- Must do two memory reads:
  - First get pointer to row array.
  - Then access element within the row.

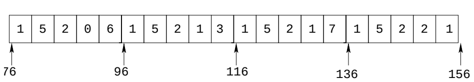
```
  salq    $2, %rsi               # 4*dig
  addq    univ(,%rdi,8),%rsi     # p = univ[index] + 4*dig
  movl    (%rsi), %eax           # return *p
  ret
```


Array Element Accesses

Nested Array

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

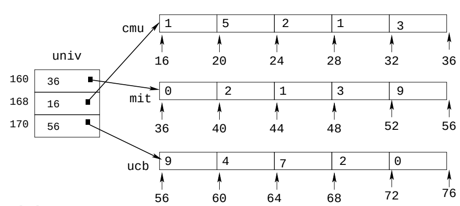
Element at
Mem[pgh+20*index+4*dig]



Multi-Level Array

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

Element at
Mem[Mem[univ+8*index]+4*dig]



Similar C references, but different address computations.

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,
             size_t i, size_t 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( size_t n, int *a,
             size_t i, size_t j ) {
  return a[IDX(n, i, j)];
}
```

Variable dimensions, implicit indexing: Now supported by gcc

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

```
  # 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     # M[a + 64*i + 4*j]
```


n x n Matrix Access

Array Elements

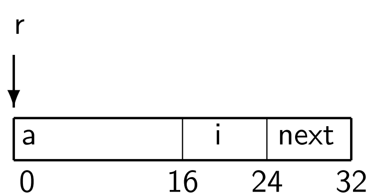
- Address $A + i * (C * K) + j * K$
- C = n, K = 4
- Must perform integer multiplication

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

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi               # n*i
  leaq    (%rsi, %rdi, 4), %rax    # a + 4*n*i
  movl    (%rax, %rcx, 4), %eax    # a + 4*n*i + 4*j
  ret
```


Structure Representation

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



Structure represented as block of memory

- Big enough to hold all the fields

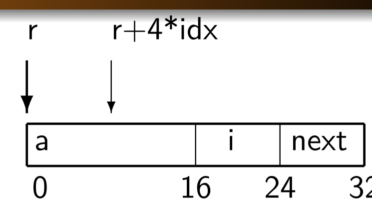
Fields ordered according to declaration

- Even if another ordering could yield a more compact representation

Compiler determines overall size and position of fields

- Machine-level program has no understanding of the structures in the source code.

Generating Pointer to Structure Member




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

```
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
```

Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as `r + 4*idx`

BTW: why does `r->i` need 8 bytes? Alignment. (Next slide set)

Aside on Structures: Arrow vs. Dot

If you have a *pointer* `r` to a structure, use `r->x` to access component `x`.

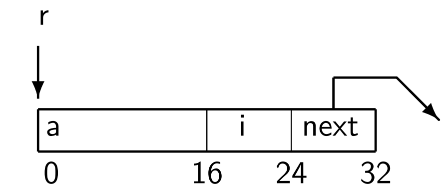
If you have the structure `s` itself, use `s.x`.

`r->x` is just syntactic sugar for `(*r).x`

Following Linked List

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

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




```
.L11:                                # loop:
   testq    %rdi, %rdi               #   Test r
   je       .L12                     #   if = 0, goto done
   movq     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]
   jmp      .L11                     #   goto loop
.L12:                                # done:
```