Computer Architecture
Machine Representation of Programs: Data

Cain Susko

Queen's University
School of Computing

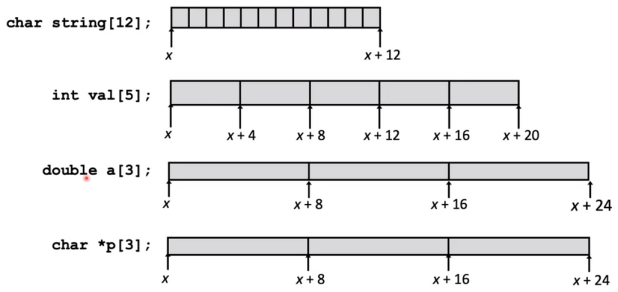
February 16, 2022

Arrays

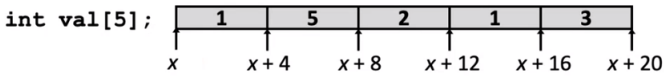
Arrays are a way of storing data of the same type:

```
T A[L].
```

This is an array A of data Type T and length L. Additional Examples are:



Arrays can be accessed using pointers like so:

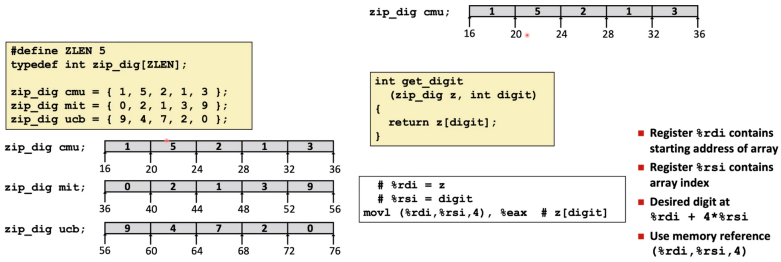


Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 i

Note: *val* is equal to the pointer at the start of the array.

1D Arrays

A comprehensive example of an array is the following:



Consider the example of a n Array in a loop: Note: this is just one of many

```
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
  rep; ret
```

forms of loops we covered in a previous lesson. Additionally, note that the *ret* instruction is used but there is no value returned (acceptable usage).

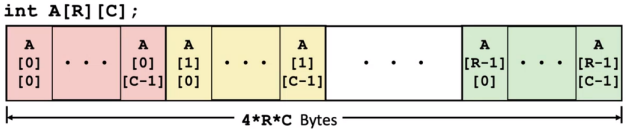
Multi-Dimensional Arrays

the declaration of a multi-dimensional or nested array is:

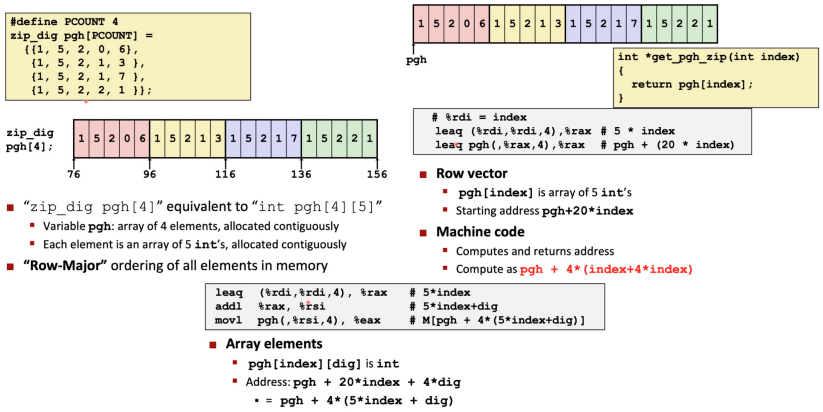
$$\texttt{T A[R][C]}.$$

where A is an array of objects with type T as well as R rows and C columns. The size of the array can be found by doing $R \times C \times k - bytes$.

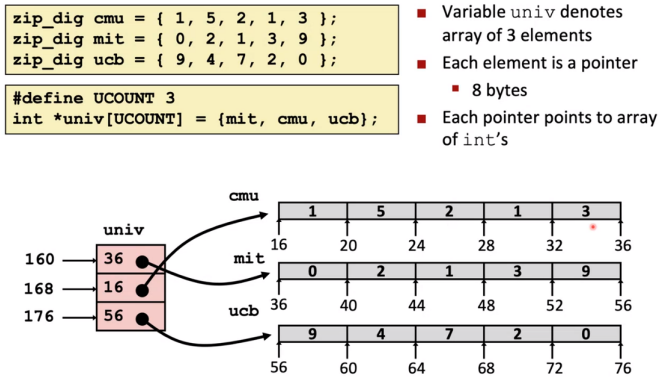
The arrangement in memory or a 2D array is row majored, meaning theyre stored row by row, linearly in memory:



Consider the following nested array example:

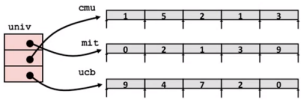


We can also make arrays of different structures, for example, the **Multi-Level Array**:



To access the elements in this type of array one could use the following procedure:

```
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

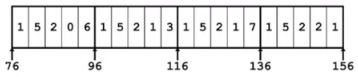
Note: because this data is not contiguous, the accessing of data may be less efficient. This can be more clearly seen in the following display:

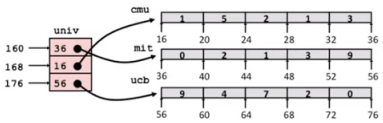
Nested array

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

Multi-level array

```
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]`**

Code in C

The representation of an $N \times N$ matrix is C must have fixed dimensions. Indexing the array can be done either with explicit indexing:

```
#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)];
}
```

Or by using 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_t j) {
  return a[i][j];
}
```

The machine code for this type of access is the following:

- **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_t 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
```