geeee

more complex data is special ways of reading bits

% eax is lower 4 bytes of % rax

% addl is lower 4 bytes of source & dest.

1-D array

starting address is array name

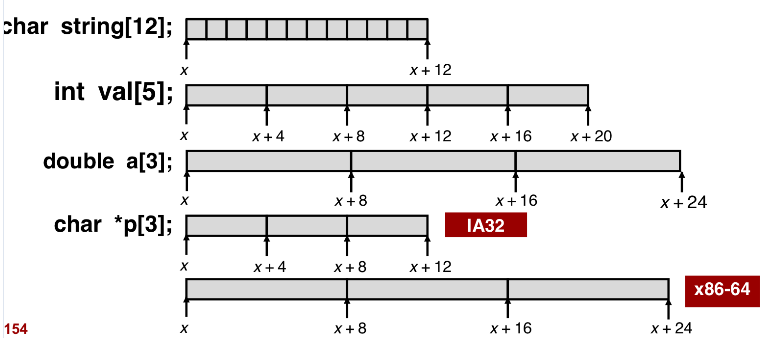
Basic principle

T **A[*L*];** e.g., int val[5]

Array of data type *T* and length *L*

Contiguously allocated region of *L* * **sizeof**(*T*) bytes

Identifier **A** can be used as a pointer to array element 0: Type *T**



154

char → 1 byte

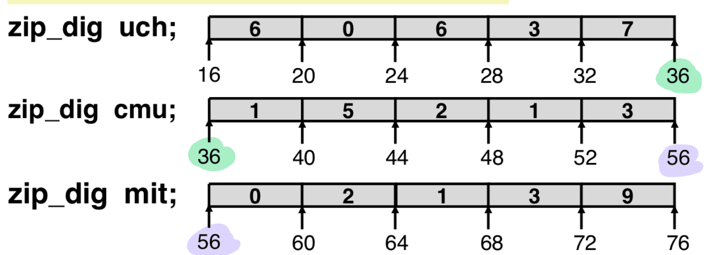
int → 4 bytes

double → 8 bytes

char* → 8 bytes (x86-64) or 4 bytes (IA32)

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

zip_dig uch = { 6, 0, 6, 3, 7 };
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
```



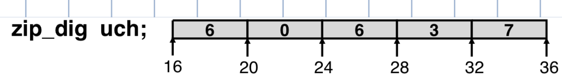
Declaration "zip_dig uch" equivalent to "int uch[ZLEN]"

These arrays were allocated in successive 20 byte blocks

Not guaranteed to happen in general

endianness changes how single byte is ordered in memory

does nothing for arrays




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

- **Register %rdi contains starting address of array**
- **Register %rsi contains array index**
- **Desired digit at 4*%rsi+%rdi**
- **Use scaled indexed memory reference (%rdi,%rsi,4)**

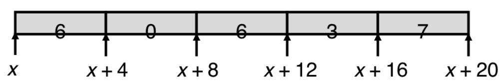
```
# %rdi = z
# %rsi = dig
  movl (%rdi,%rsi,4),%eax   # z[dig]
```

%eax: lower 32 bits of **%rax**

movl automatically zeros higher 32 bits

4

```
int val[5];
```



Reference	Type	Value
val[4]	int	7
val	int *	*x*
val+1	int *	*x* + 4
&val[2]	int *	*x* + 8
val[5]	int	??
*(val+1)	int	0
val + *i*	int *	*x* + 4 *i*

Multidimensional arrays

① Nested arrays

Multidimensional (Nested) Arrays

Declaration

T **A[*R*][*C*];**

2D array of data type *T*

R rows, *C* columns

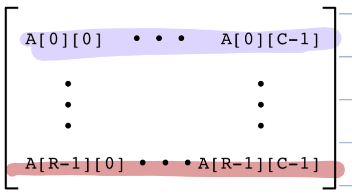
Type *T* element requires *K* bytes

Array Size

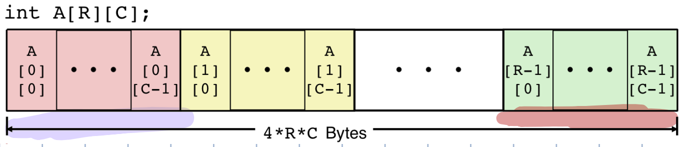
R * *C* * *K* bytes

Arrangement

Row-Major Ordering



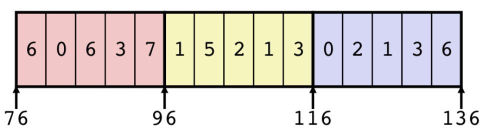

```
int A[R][C];
```



row major ordering

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

```
zip_dig
univ[3];
```



Row Vectors

A[i] is array of *C* elements

Each element of type *T* requires *K* bytes

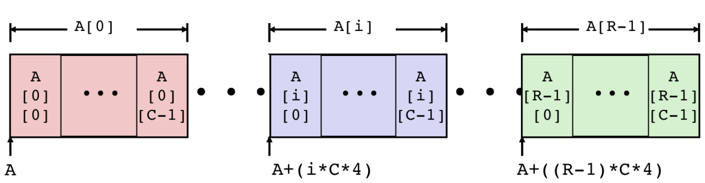
Starting address A + *i* * (*C* * *K*)

C elements

i row

k bytes

```
int A[R][C];
```




```
# %rdi = index, %rsi = digit, %rdx = univ
   leaq   (%rdi,%rdi,4), %rax # 5*index
   addq   %rax, %rsi    # 5*index+digit
movl   %rdx(,%rsi,4), %eax # Mem[univ + 4*(5*index+digit)]
```

① Multi level arrays

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uch = { 6, 0, 6, 3, 7 };
zip_dig mit = { 0, 2, 1, 3, 6 };
```

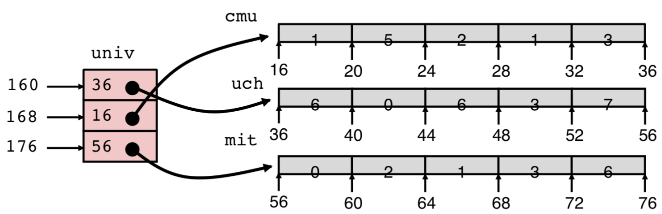
```
#define UCOUNT 3
int *univ[UCOUNT] = {uch, cmu, mit};
```

Variable univ denotes array of 3 elements

Each element is a pointer

8 bytes → *array of pointers*

Each pointer points to array of int's



each pointer has 8bytes

Nested array

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

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

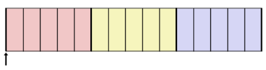
Multi-level array

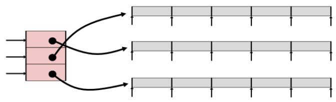
```
#define UCOUNT 3
int *univ[UCOUNT] = {uch, cmu, mit};
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uch = { 6, 0, 6, 3, 7 };
zip_dig mit = { 0, 2, 1, 3, 6 };

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

same function

different memory storage





Accesses look similar in C, but address computations very different:

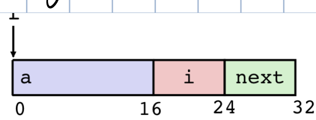
Mem[univ+20*index+4*digit] **Mem[Mem[univ+8*index]+4*digit]**

Structures

predefined sequence of elements

array w/variable length

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



size_t is unsigned integer in C (8 bytes on 64-bit machines)

Structure represented as block of memory

- **Big enough to hold all of the fields**

Fields ordered according to declaration

- **Even if another ordering could yield a more compact representation**

Compiler determines overall size + positions of fields

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

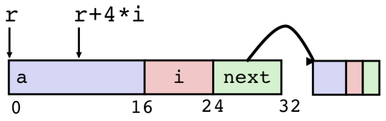
all assembly cares about:

relative memory address to starting memory of structure

bytes from start struct to element

linked lists

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




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

```
.L11:                          # loop:
  movslq  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]
  testq   %rdi, %rdi           # Test r
  jne     .L11                 # if !=0 goto loop
```

Register	Value
%rdi	r
%rsi	val

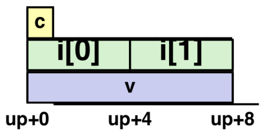
addresses need to be multiple of largest element

Union only needs 1 field at a time → allocate by largest element

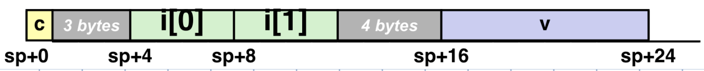
Allocate according to largest element

Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Union Example

```
union {
  unsigned char C[8];
  unsigned int I[2];
} dw;
```

```
union dw arg;
for (int i = 0; i < 8; i++){
   arg.C[i] = i;
}
printf("%x\n", arg.I[0]);
printf("%x\n", arg.I[1]);
```

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
??				??			
I[0]				I[1]			

Little endian

0x00 LSB	0x01	0x02	0x03 MSB	0x04	0x05	0x06	0x07
0x03020100				0x07060504			

Big endian

0x00 MSB	0x01	0x02	0x03 LSB	0x04	0x05	0x06	0x07
0x00010203				0x04050607			