CSCI-UA.0201

Computer Systems Organization

Machine Level – Manipulating Data

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

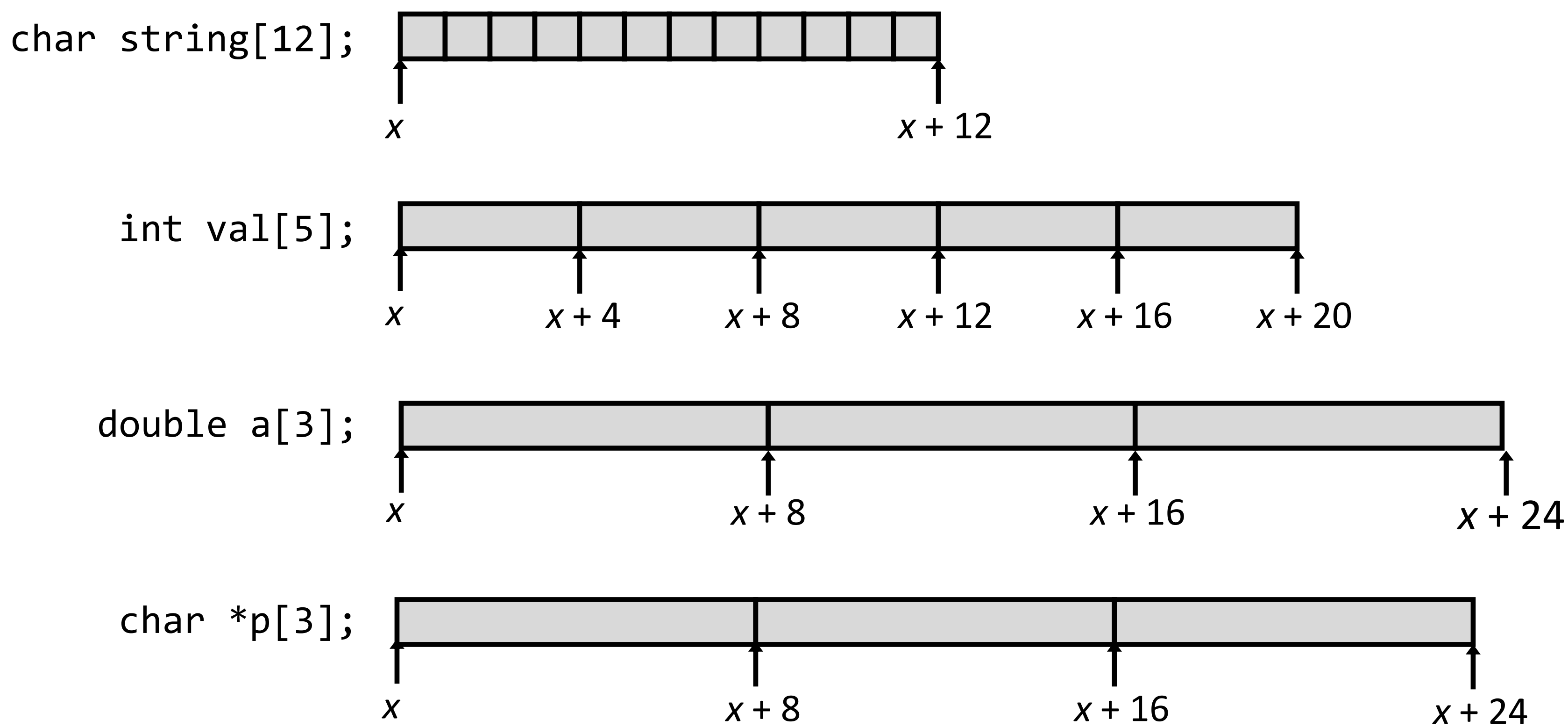
Manipulating Data

How are data structures, like arrays, presented and manipulated in assembly?

Array Allocation

- Basic Principle

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



Array Access

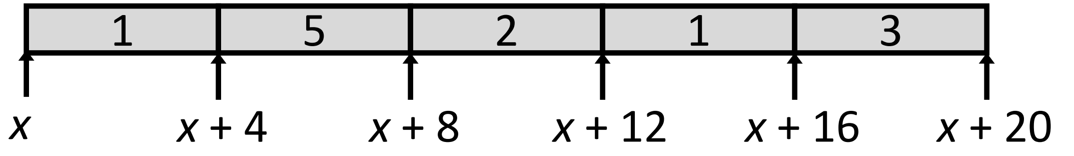
- Basic Principle

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

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

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

`int val[5];`

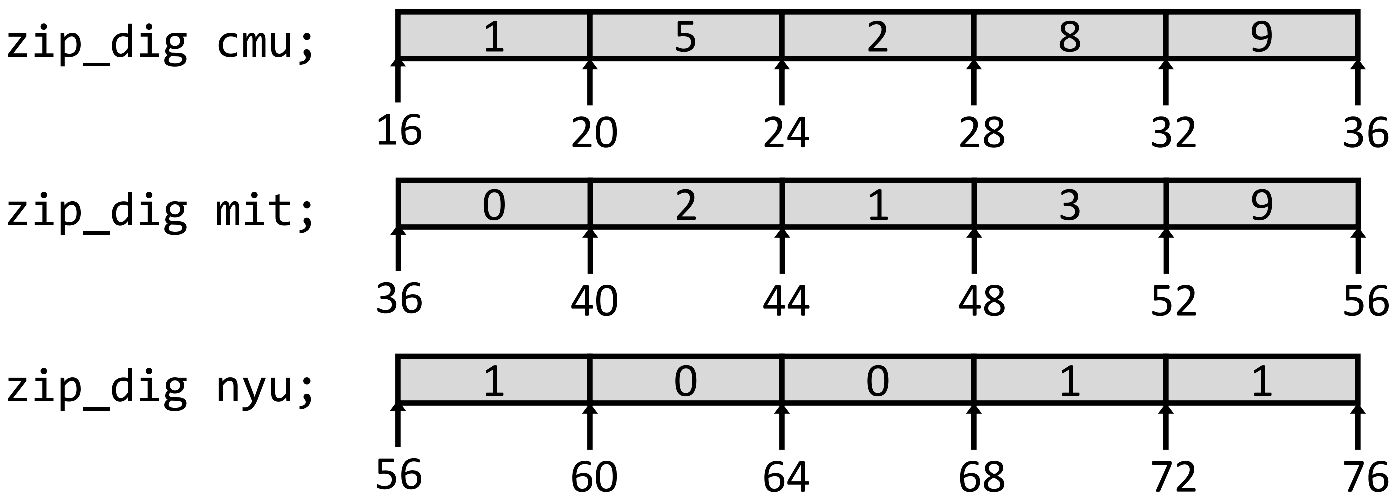


- Reference Type Value

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

Array Example

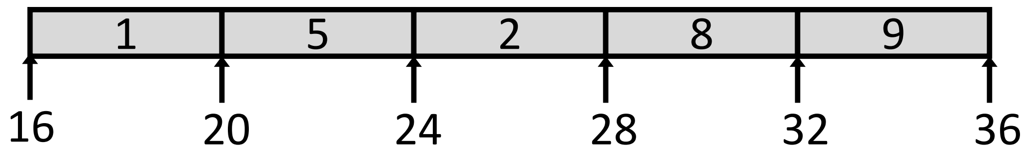
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 8, 9 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig nyu = { 1, 0, 0, 1, 1 };
```



- Declaration "`zip_dig nyu`" equivalent to "`int nyu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

zip_dig cmu;




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

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

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

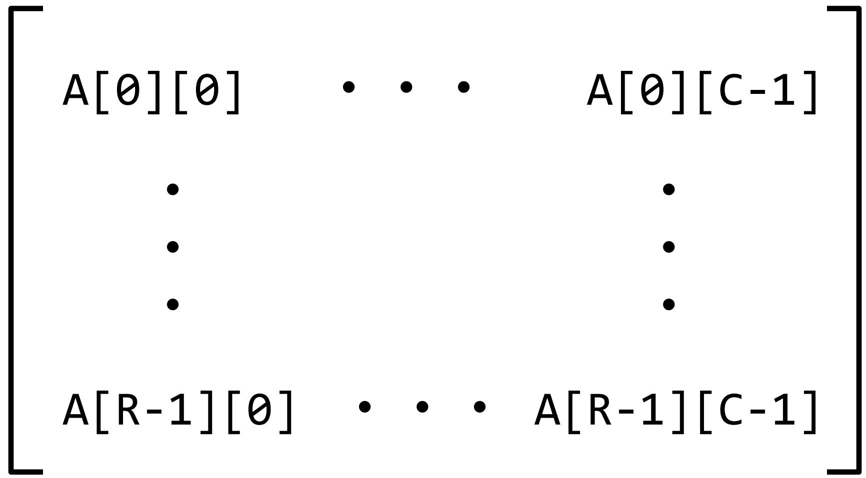
Array Loop Example

```c
void zincr(int * z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

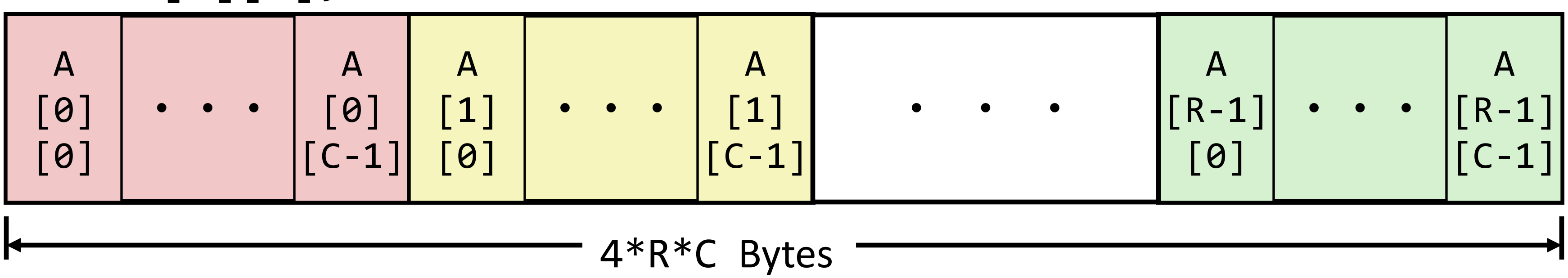
```
  # %rdi = z
  # ZLEN is 5
  movl    $0, %eax              #   i = 0
  jmp     .L3                   #   goto middle
.L4:                            # loop:
  addl    $1, (%rdi,%rax,4)     #   z[i]++
  addl    $1, %eax              #   i++
.L3:                            # middle
  cmpl    $4, %eax              #   i:4
  jbe     .L4                   #   if <=, goto loop
  ret
```

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 in memory
  - Row-Major Ordering

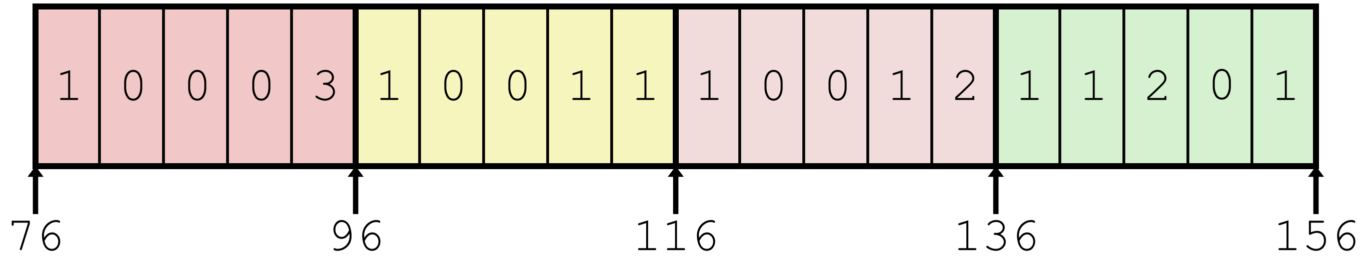



```
int A[R][C];
```



Nested Array Example

```
int nyu[4][5] =
  {{1, 0, 0, 0, 3},
   {1, 0, 0, 1, 1 },
   {1, 0, 0, 1, 2 },
   {1, 1, 2, 0, 1 }};
```



- Variable **nyu**: array of 4 elements, allocated contiguously
- Each element is an array of 5 **`int`**'s, allocated contiguously

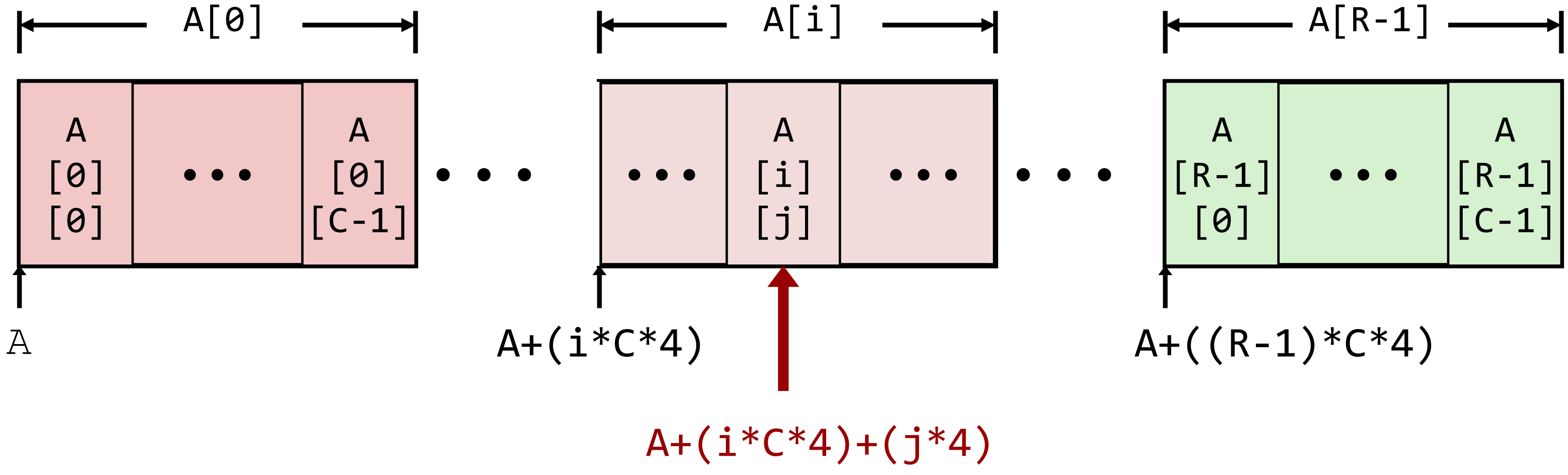
- "Row-Major" ordering of all elements in memory

Nested Array Element Access

- Array Elements
  - address of **`A[i][j]`**:

    Address $\mathbf{A} + i * (C * K) + j * K = \mathbf{A} + (i * C + j) * K$

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

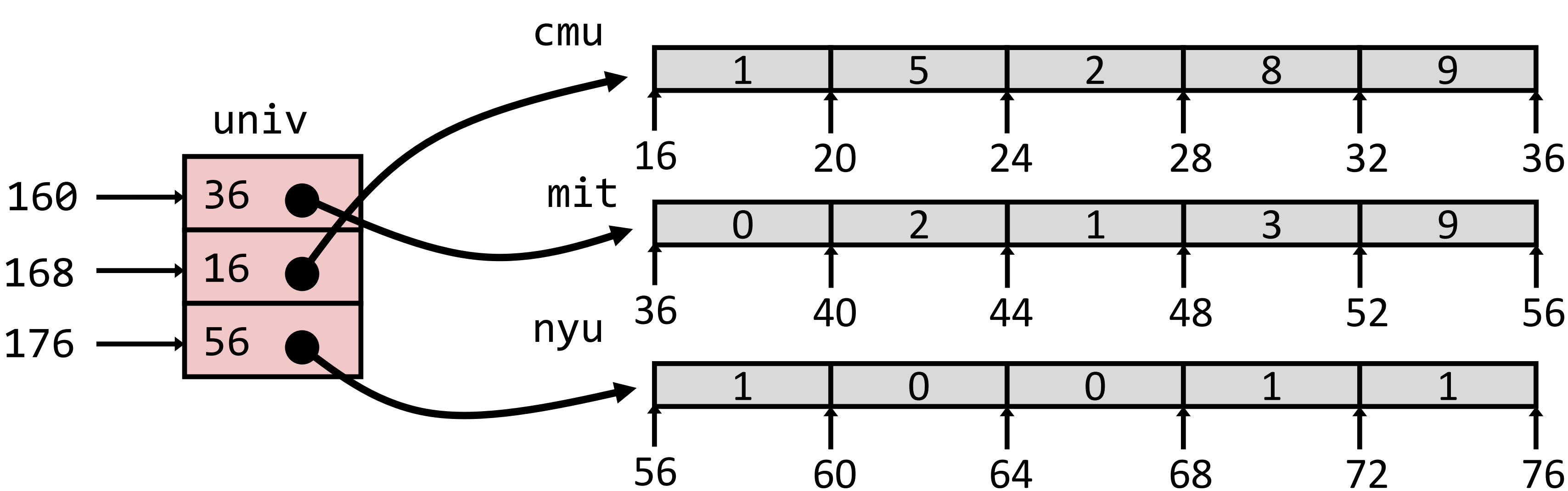


Multi-Level Array Example

```
int cmu[5] = { 1, 5, 2, 8, 9 };
int mit[5] = { 0, 2, 1, 3, 9 };
int nyu[5] = { 1, 0, 0, 1, 1 };
```

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

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



Element Access in Multi-Level Array

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

```
%rdi = index
%rsi = digit
```

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

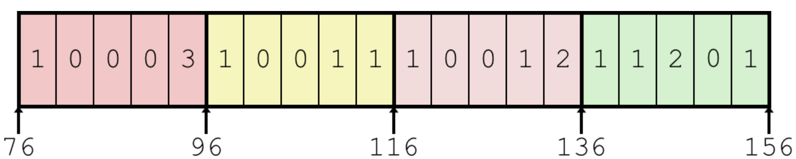
Array Element Accesses

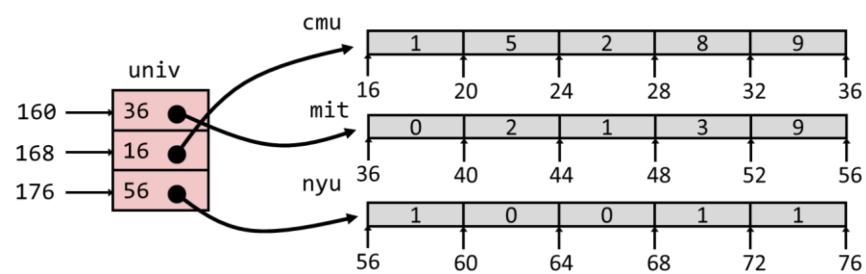
Nested array

```
int get_nyu_digit
  (size_t index, size_t digit)
{
  return nyu[index][digit];
}
```

Multi-level array

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





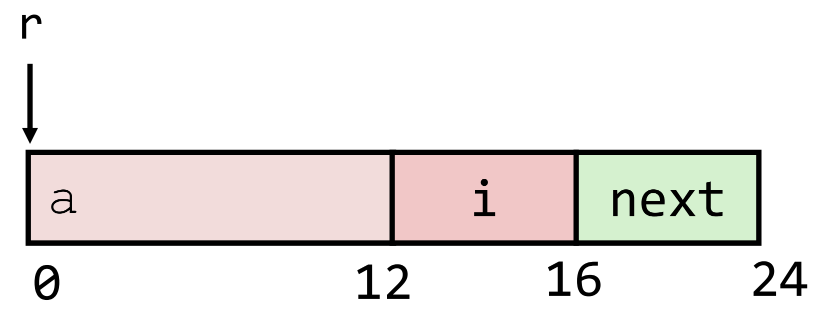
Accesses looks similar in C, but address computations very different:

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

How about structures?

Structure Representation

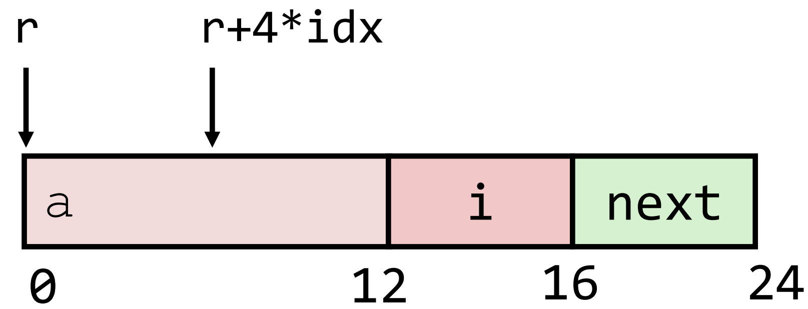
```
struct rec {
    int a[3];
    size_t i;
    struct rec *next;
};
```



- 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**

Generating Pointer to Structure Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as **`r + 4*idx`**

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

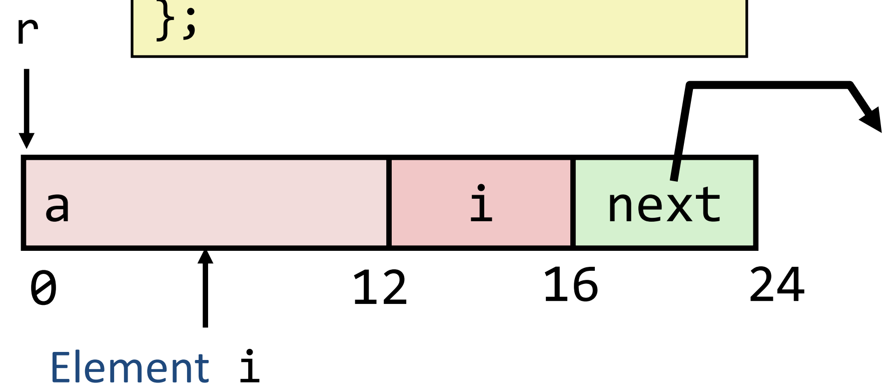
```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

Following Linked List

- C Code

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

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



Register	Value
%rdi	r
%rsi	val

```
.L3:                             # loop:
  movslq  12(%rdi), %rax         #   i = M[r+12]
  movl    %esi, (%rdi,%rax,4)    #   M[r+4*i] = val
  movq    16(%rdi), %rdi         #   r = M[r+16]
  testq   %rdi, %rdi             #   Test r
  jne     .L3                    #   if !=0 goto loop
```

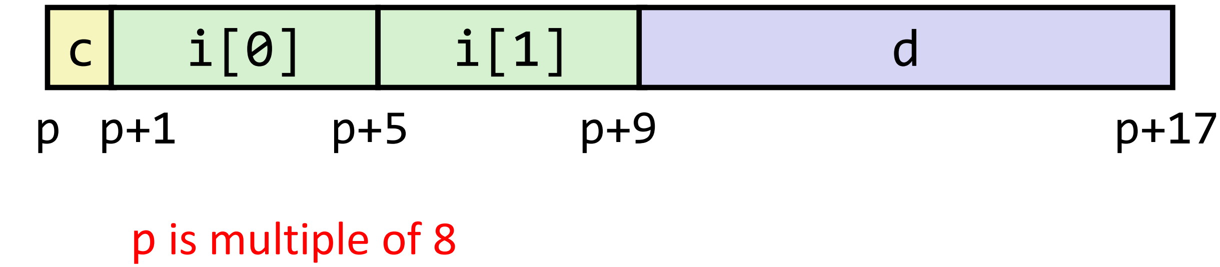
Alignment

Alignment Principles

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries (i.e. 8 bytes boundaries)
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

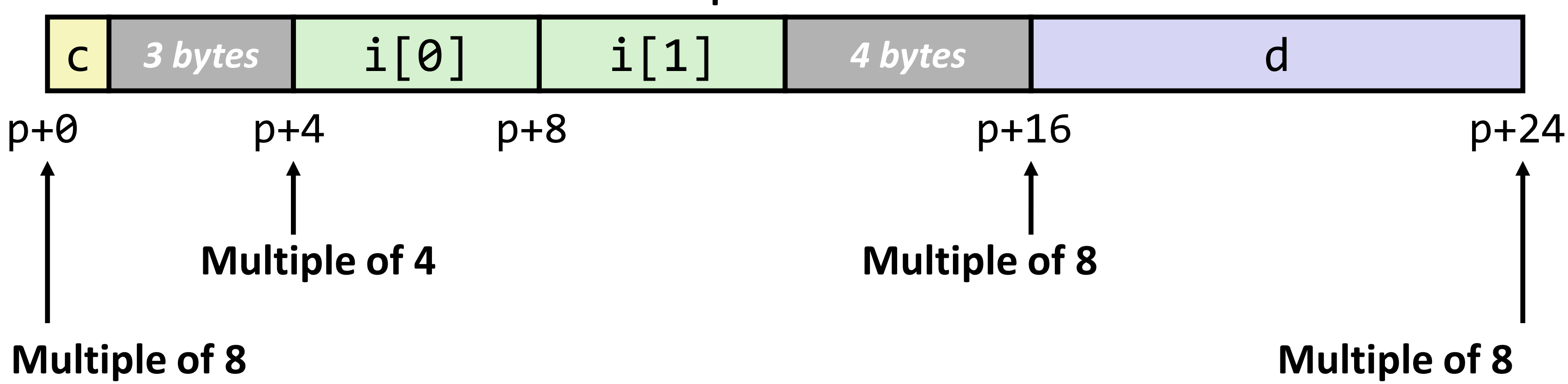
Structures & Alignment

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double d;
} *p;
```

- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



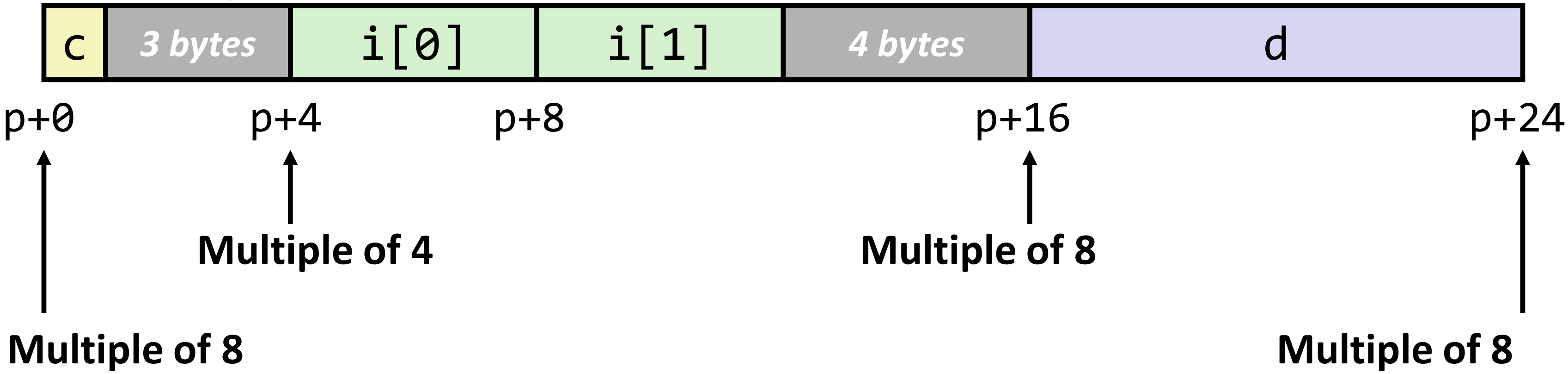
Specific Cases of Alignment (x86-64)

- 1 byte: `char`, …
  - no restrictions on address
- 2 bytes: `short`, …
  - address must be multiple of 2
- 4 bytes: `int, float`, …
  - address must be multiple of 4
- 8 bytes: `double, long, char *`, …
  - address must be multiple of 8
- 16 bytes: `long double` (GCC on Linux)
  - address must be multiple of 16

How about structures?

```
struct S1 {
  char c;
  int i[2];
  double d;
} *p;
```

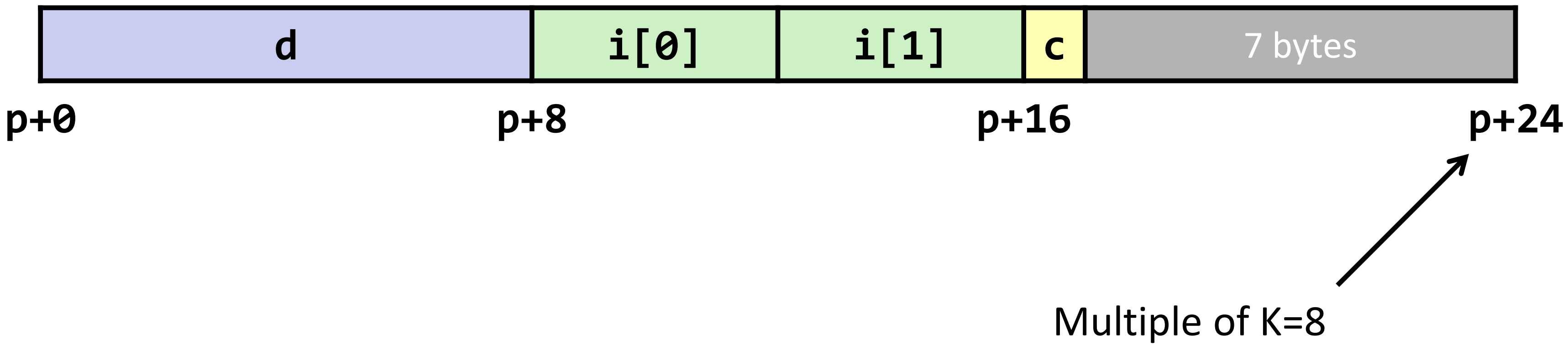
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- Example:
  - K = 8, due to **`double`** element



Meeting Overall Alignment Requirement

```
struct S2 {
  double d;
  int i[2];
  char c;
} *p;
```

- For largest alignment requirement K
- Overall structure must be multiple of K



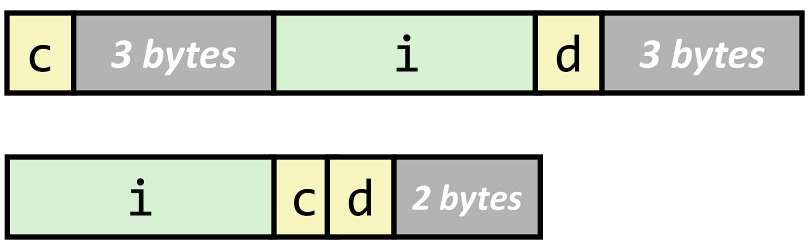
Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)

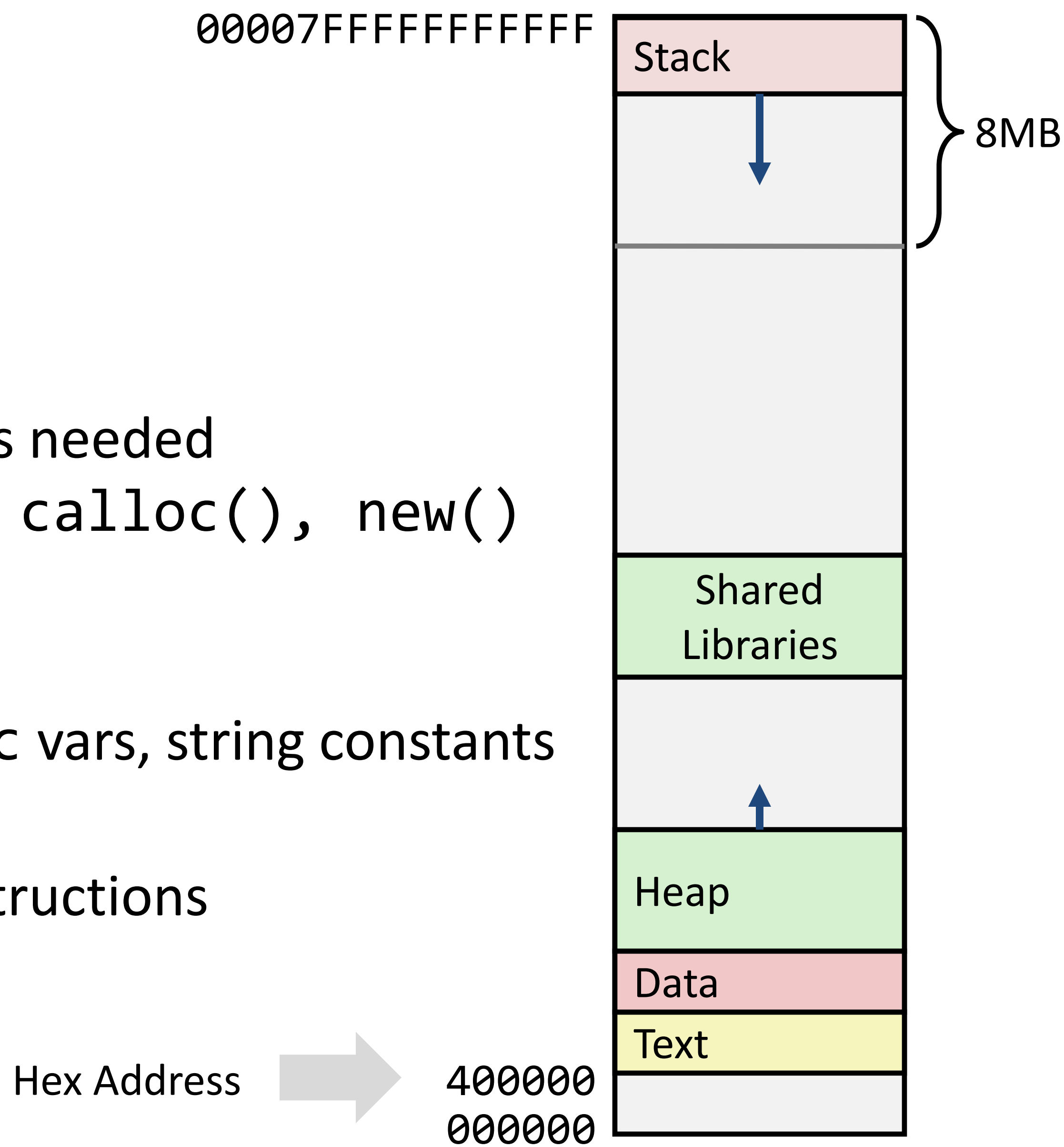


Memory Layout Revisited

x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack
  - 8MB default limit
- Heap
  - Dynamically allocated as needed
  - When call `malloc(), calloc(), new()`
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



Conclusions

- We have not covered everything in x86-64, just gave you a glimpse and a feel for it.
- Compiler does more than blindly translating your high-level language (HLL) code:
  - It manages the stack / register allocation.
  - It translates the sophisticated data structure access to assembly
  - It optimizes your code
- No matter how sophisticated your HLL language code, it will be translated to assembly with 16 registers and basic data types!