[CSED211] Introduction to Computer Software Systems

Lecture 7: Data

Prof. Jisung Park

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.10.11

Lecture Agenda

- Arrays
  - One-Dimensional
  - Multi-Dimensional (Nested)
  - Multi-Level

- Structures
  - Allocation
  - Access
  - Alignment

- Floating Point

Basic Data Types

- Integral
  - Stored and operated on in general (integer) registers
  - Signed or unsigned depending on instructions used

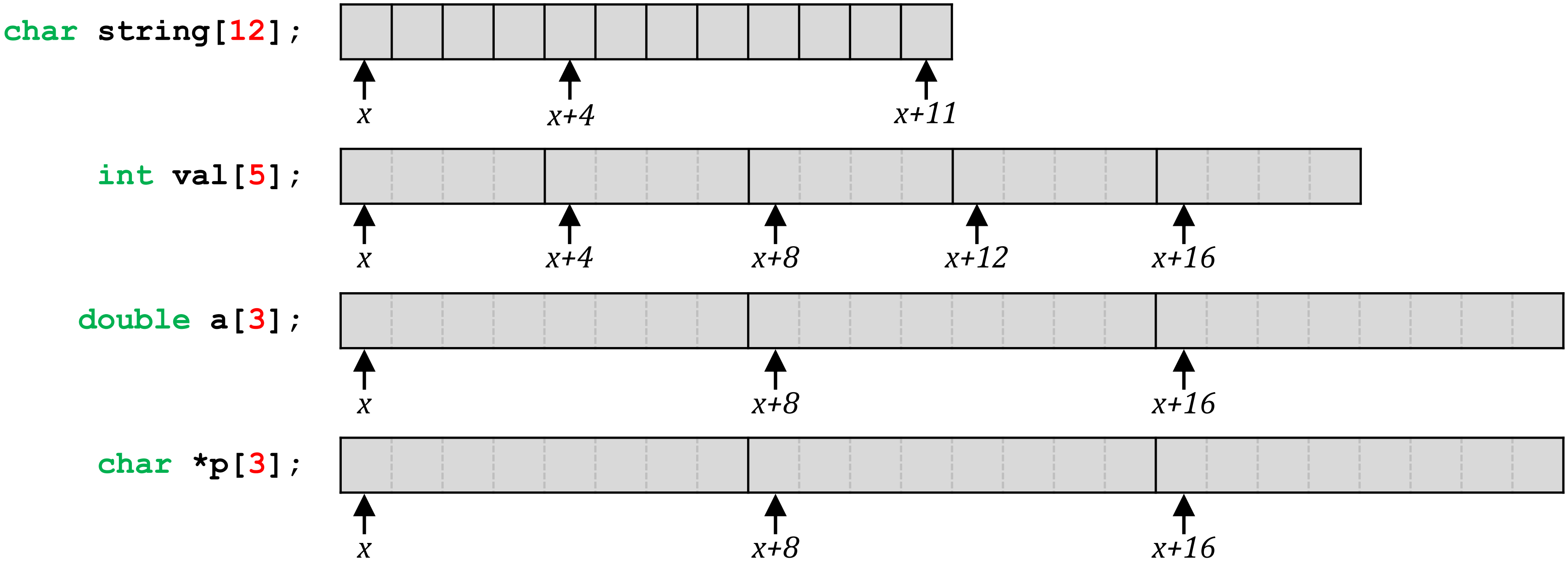
Intel	ASM	Bytes	C
Byte	`b`	1	`[unsigned] char`
Word	`w`	2	`[unsigned] short`
Double word	`l`	4	`[unsigned] int`
Quad word	`q`	8	`[unsigned] long int` (x86-64)

- Floating point
  - Stored and 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

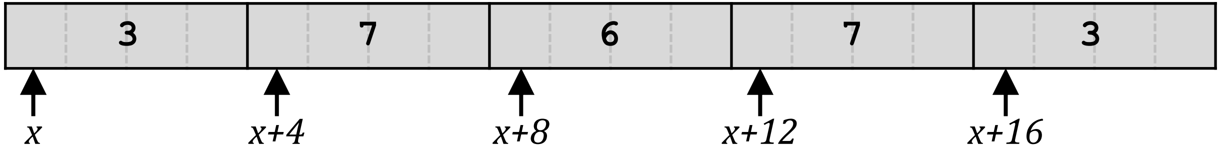
- Basic Principle: `T N[L];`
  - Array of data type `T` and length `L` named as `N`
  - Contiguously allocated region of (`L * sizeof(T)`) bytes



Array Access

- Basic Principle: `T N[L];`
  - Array of data type `T` and length `L` named as `N`
  - Contiguously allocated region of (`L * sizeof(T)`) bytes

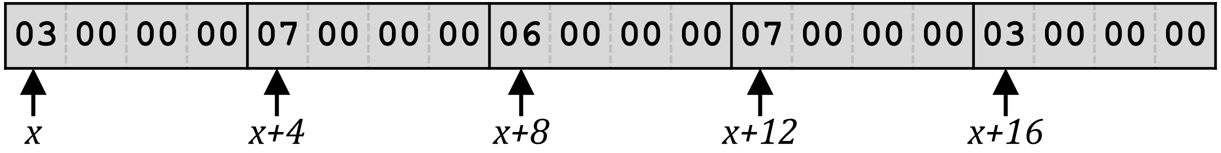
`int val[5];`



Array Access

- Basic Principle: `T N[L];`
  - Array of data type `T` and length `L` named as `N`
  - Contiguously allocated region of (`L * sizeof(T)`) bytes

`int val[5];`



Reference	Type	Value
`val[4]`	`int`	`3`
`val[5]`	`int`	`??`
`val`	`int*`	`x`
`val+1`	`int*`	`x + 4`
`&val[2]`	`int*`	`x + 8`
`*(val+1)`	`int`	`7`
`val+i`	`int*`	`x + 4*i`

Array Example

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

zip_dig postech = {3, 7, 6, 7, 3};
zip_dig snu     = {0, 8, 8, 2, 0};
zip_dig kaist   = {3, 4, 1, 4, 1};
```

- Declaration '`zip_dig postech`' equivalent to '`int postech[5]`'
- Example arrays are allocated in successive 20-byte blocks
  - Not guaranteed to happen in general

`zip_dig postech;`

03	00	00	00	07	00	00	00	06	00	00	00	07	00	00	00	03	00	00	00
16				20				24				28				32			

`zip_dig snu;`

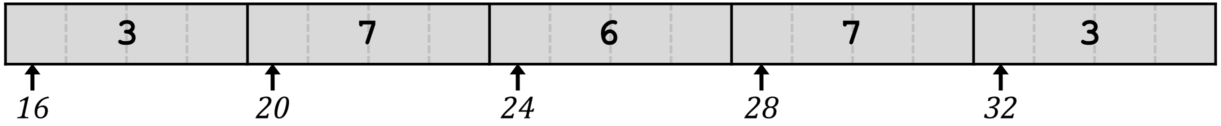
00	00	00	00	08	00	00	00	08	00	00	00	02	00	00	00	00	00	00	00
36				40				44				48				52			

`zip_dig kaist;`

03	00	00	00	04	00	00	00	01	00	00	00	04	00	00	00	01	00	00	00
56				60				64				68				72			

Array Accessing Example

`zip_dig postech;`




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

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

- Register **`%rdi`** contains the target array's starting address
- Register **`%rsi`** contains the target array index
- Desired digit at **`%rdi+4×%rsi`**, i.e., **`(%rdi,%rsi,4)`**

Array Loop Example

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

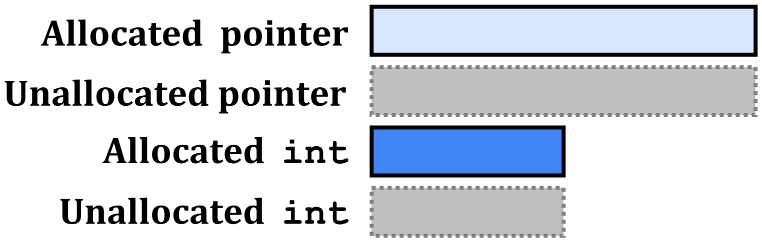
```
  # rdi = z
  movl    $0, %eax          #   %eax = i
  jmp     .L3
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   compare i and 4
  jbe     .L4               #   if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

Understanding Pointers & Arrays#1

Declaration	A1, A2			*A1, *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`						
`int *A2`						

- Comp: can be compiled (Y/N)
- Bad: possible bad pointer reference (Y/N)
- Size: value returned by `sizeof`

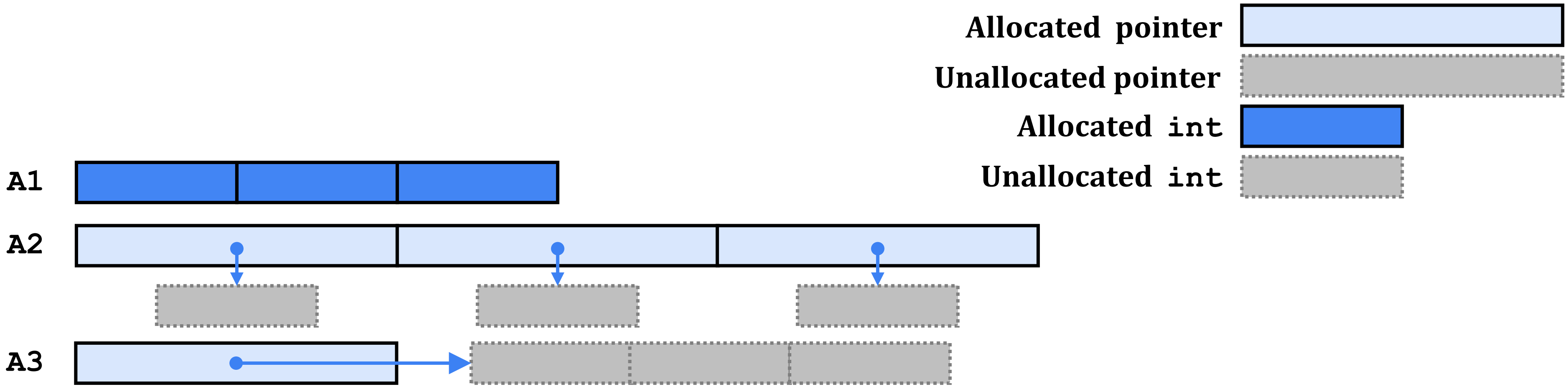
Understanding Pointers & Arrays#1



Declaration	A1, A2			*A1, *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4

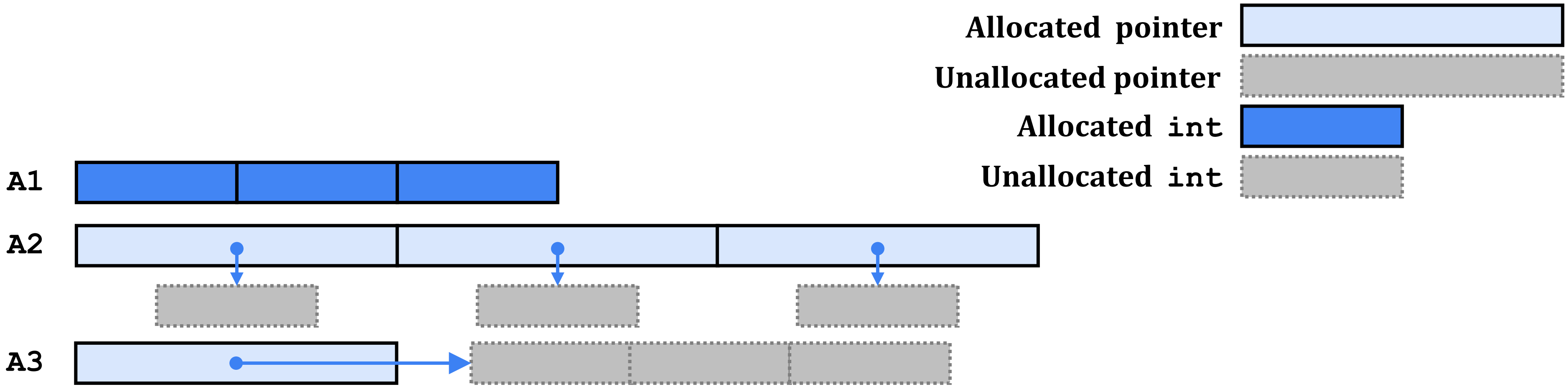
- Comp: can be compiled (Y/N)
- Bad: possible bad pointer reference (Y/N)
- Size: value returned by `sizeof`

Understanding Pointers & Arrays#2



Declaration	An			*An			**An		
	Comp	Bad	Size	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									

Understanding Pointers & Arrays#2



Declaration	An			*An			**An		
	Comp	Bad	Size	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	-	-
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4

Multidimensional (Nested) Arrays

- Declaration: `T N[R][C];`
  - A 2D array of data type `T`
  - `R` rows and `C` columns
- Array Size
  - `R*C*k` bytes
  - Where `sizeof(T)=k`
- Arrangement: Row-major ordering

`int A[R][C];`

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

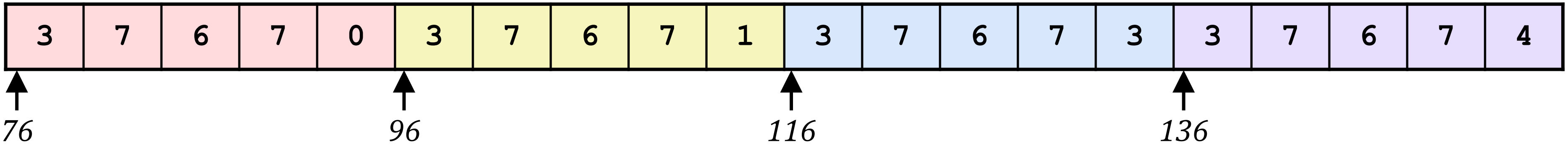
A[0] [0]	•••	A[0] [C–1]	A[1] [0]	•••	A[1] [C–1]	• • •	A[R–1] [0]	•••	A[R–1] [C–1]

`4*R*C` Bytes

Nested Array Example

```
#define PCOUNT 4
zip_dig pohang[PCOUNT] = {{3, 7, 6, 7, 0},
                          {3, 7, 6, 7, 1},
                          {3, 7, 6, 7, 3},
                          {3, 7, 6, 7, 4}};
```

`zip_dig pohang[4];`

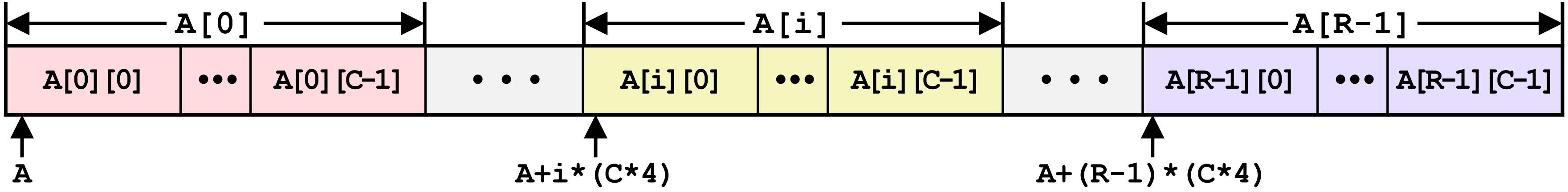


- '**`zip_dig pohang[4]`**' equivalent to '**`int pohang[4][5]`**'
  - Variable **`pohang`**: an array of 4 elements allocated contiguously
  - Each element is an array of 5 **`int`**'s also allocated contiguously
- Row-major ordering of all elements guaranteed

Nested Array Row Access

- Row vectors `T N[R][C]`
  - `N[i]` is an array of `C` elements
  - Each element of type `T` requires `k` bytes
  - Starting address `N + i * (C * k)`

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



Nested Array Row Access Code

```
zip_dig pohang[4];
```

3	7	6	7	0	3	7	6	7	1	3	7	6	7	3	3	7	6	7	4
76					*96*					*116*					*136*				

```c
int *get_pohang_zip(int index){
  return pohang[index];
}
```

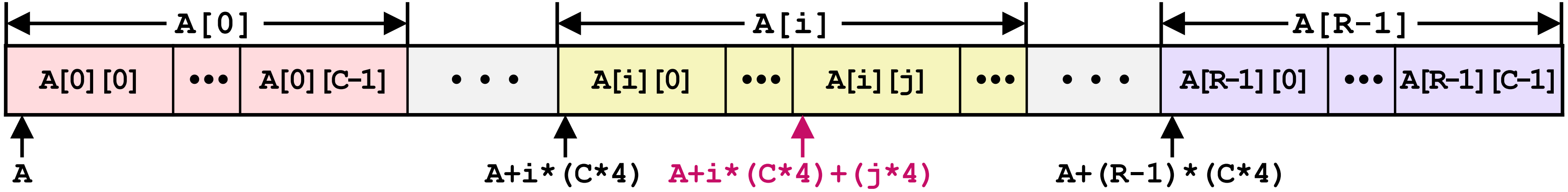
```
# %rdi = index
leaq (%rdi,%rdi,4),%rax      # 5 * index
leaq pohang(,%rax,4),%rax    # pohang+(20*index)
ret
```

- Row vector **pohang**
  - **pohang[index]** is an array of 5 **int**'s
  - Starting address **pgh+20*index**
- Machine Code
  - Computes and returns address
  - Computes as **pgh + 4*(index+4*index)**

Nested Array Element Access

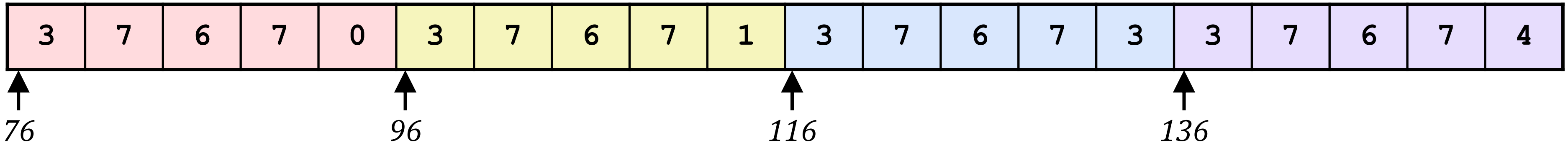
- Array elements
  - `N[i][j]` is an element of type `T` that requires `k` bytes
  - Address `N+i*(C*K)+j*k = N+(i*C+j)*K`

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



Nested Array Element Access Code

```
zip_dig pohang[4];
```




```
int *get_pohang_dig(int index, int dig){
  return pohang[index][dig];
}
```

```
leaq     (%rdi,%rdi,4), %rax      # 5*index
addl     %rax, %rsi               # 5*index+dig
movl     pohang(,%rsi,4), %eax    # M[pohang+4*(5*index+dig)]
ret
```

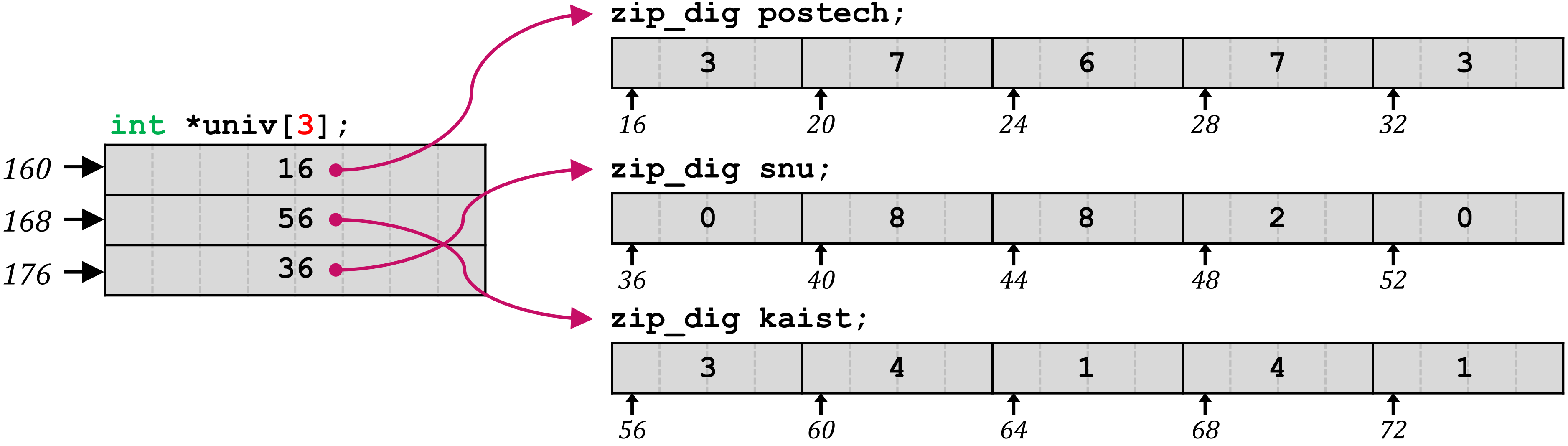
- Array Elements
  - `pohang[index][dig]` is `int`
  - Address: `pohang+20*index+4*dig = pohoang+4*(5*index+dig)`

Multi-Level Array Example

```
zip_dig postech = {3, 7, 6, 7, 3};
zip_dig snu     = {0, 8, 8, 2, 0};
zip_dig kaist   = {3, 4, 1, 4, 1};

#define UCOUNT 3
int *univ[UCOUNT] = {postech, kaist, snu};
```

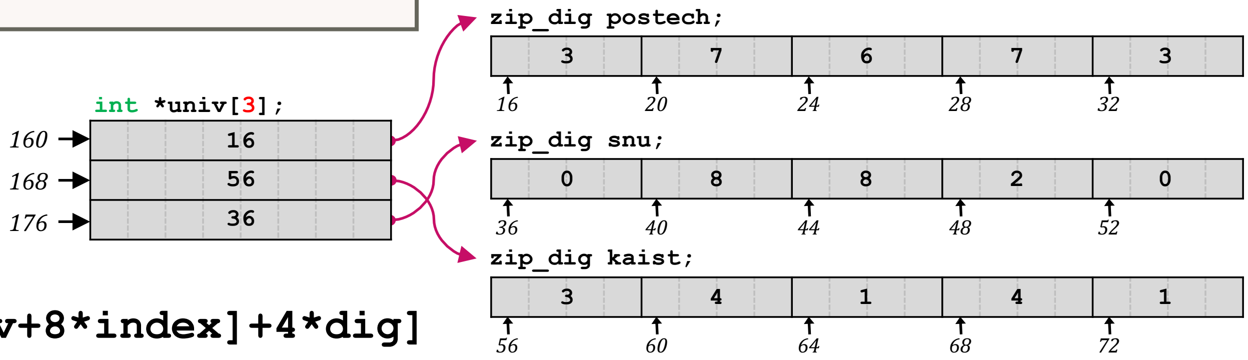
- Variable **`univ`** is an array of 3 pointer elements (8 bytes)
- Each points to an array of 5 `int`'s



Element Access in Multi-Level Array

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

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



- Computation
  - Access **Mem[Mem[univ+8*index]+4*dig]**
  - Must do two memory reads
    - First to get the pointer to the target row array
    - Second to access the target element within the array

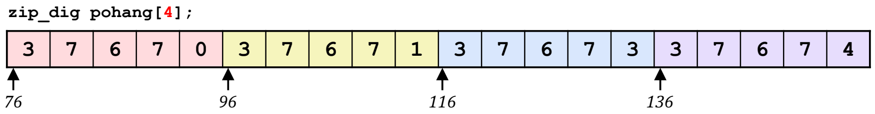
Array Element Accesses

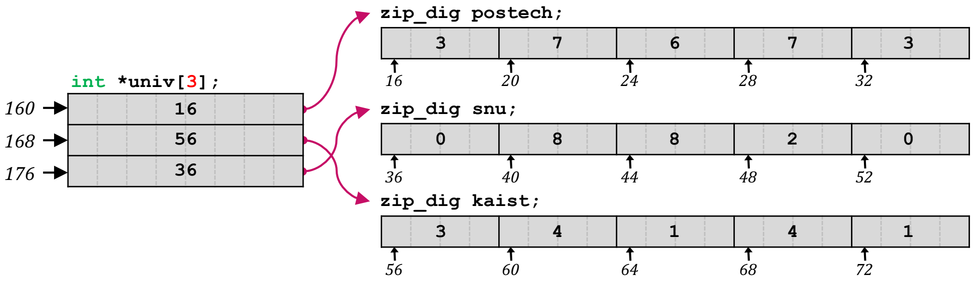
Nested Array

```
int *get_pohang_dig(int index, int dig){
  return pohang[index][dig];
}
```

Multi-Level Array

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





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

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

`Mem[Mem[univ+8*index]+4*dig]`

N×N Matrix Code

- Fixed dimensions
  - A known 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(size_t 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(size_t n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

16×16 Matrix Access

- **Array elements**
  - Address **N+i*(C*K)+j*K**
  - **C** = 16, **k** = 4

```
#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];
}
```

```
# 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]
ret
```

n×n Matrix Access

- **Array elements**
  - Address **N+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], int i, int 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
```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pohang[PCOUNT] = {{3, 7, 6, 7, 0},
                              {3, 7, 6, 7, 1},
                              {3, 7, 6, 7, 3},
                              {3, 7, 6, 7, 4}};
    int *linear_zip = (int *) pohang;
    int *zip2 = (int *) pohang[2];
    int result = pohang[0][0] +
                 linear_zip[7] +
                 *(linear_zip + 8) +
                 zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

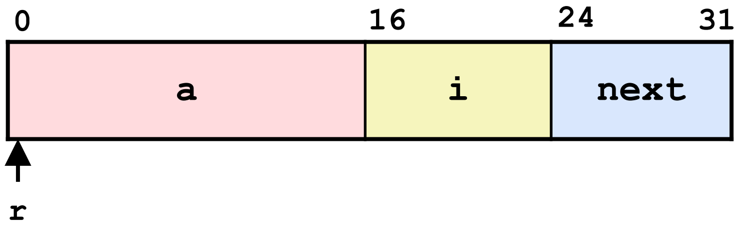
```
$ ./array
result: 23
```

Lecture Agenda

- Arrays
  - One-Dimensional
  - Multi-Dimensional (Nested)
  - Multi-Level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

Structure Representation

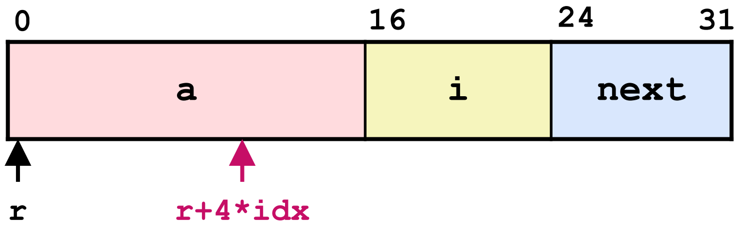
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as a memory block
  - Big enough to hold all the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines the overall size and 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[4];
    size_t i;
    struct rec *next;
};
```



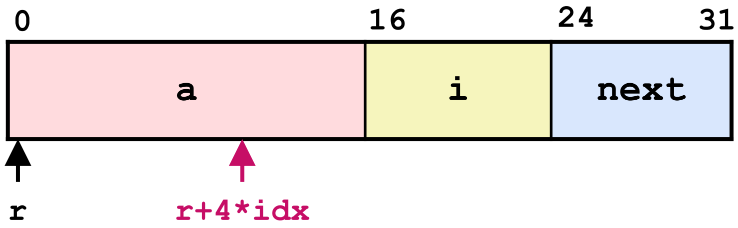
- Generating pointer to array element
  - Offset of each structure member determined at compile time
  - Compute as **`r+4*idx`**

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

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

Generating Pointer to Structure Member

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



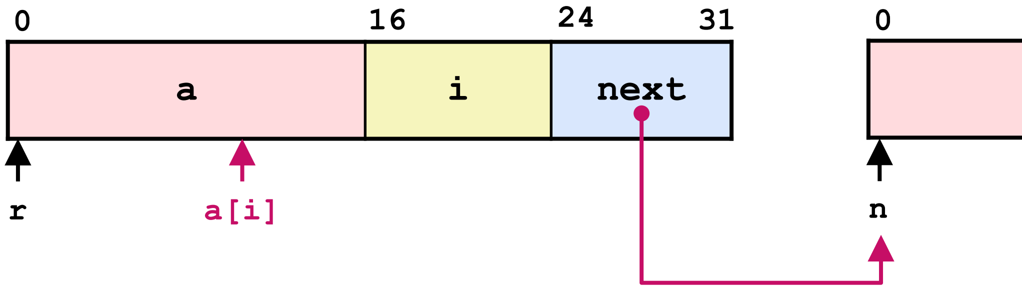

```
int *get_ap(struct rec *r, size_t idx){
  return &r->a[idx];
}
```

```
# 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`**

Generating Pointer to Structure Member

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

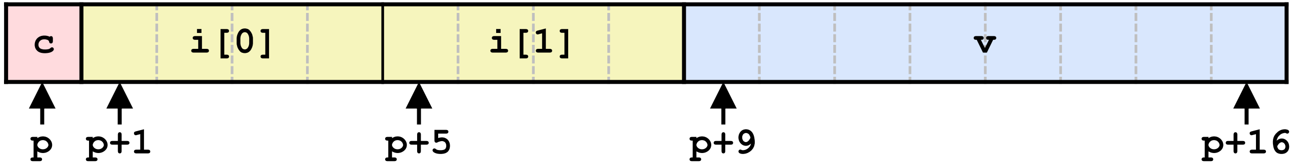
```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = Mem[r+16]
  movl    %esi, (%rdi,%rax,4)   #   Mem[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = Mem[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

Register	Use(s)
%rdi	r
%rsi	val

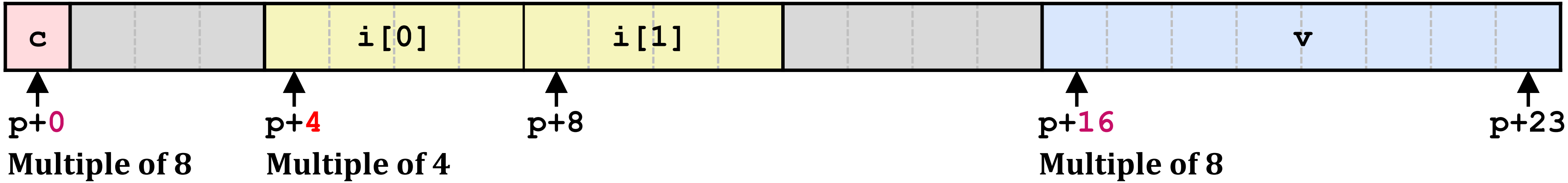
Structures & Alignment

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

- Unaligned Data



- Aligned data
  - Address must be multiple of B if primitive data type requires B bytes



Alignment Principles

- Aligned data
  - Address must be multiple of *B* if primitive data type requires *B* bytes
  - 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 data that spans quad-word boundaries
    - Virtual memory trickier when data spans 2 pages

- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

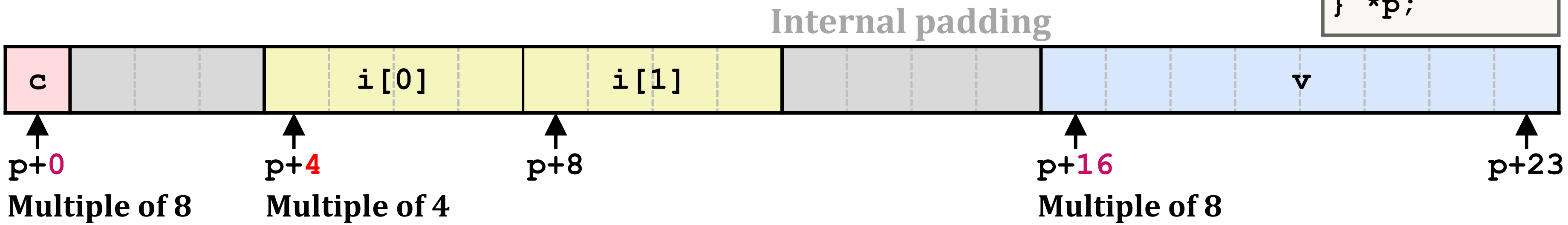
Specific Cases of Alignment (x86-64)

- 1 byte: `char`
  - No restrictions on address
- 2 bytes: `short`
  - Lowest bit of address must be 0_2
- 4 bytes: `int`, `float`, …
  - Lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `long`, `char *`, …
  - Lowest 3 bits of address must be 000_2
- 16 bytes: `long double` (gcc on Linux)
  - Lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

- 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

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

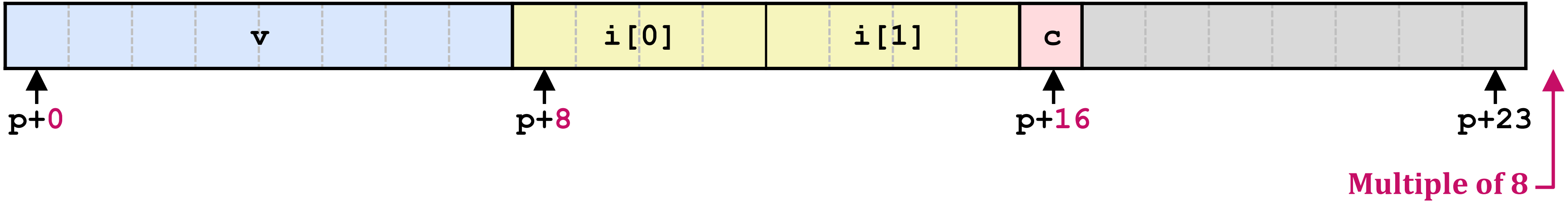


Meeting Overall Alignment Requirement

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

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

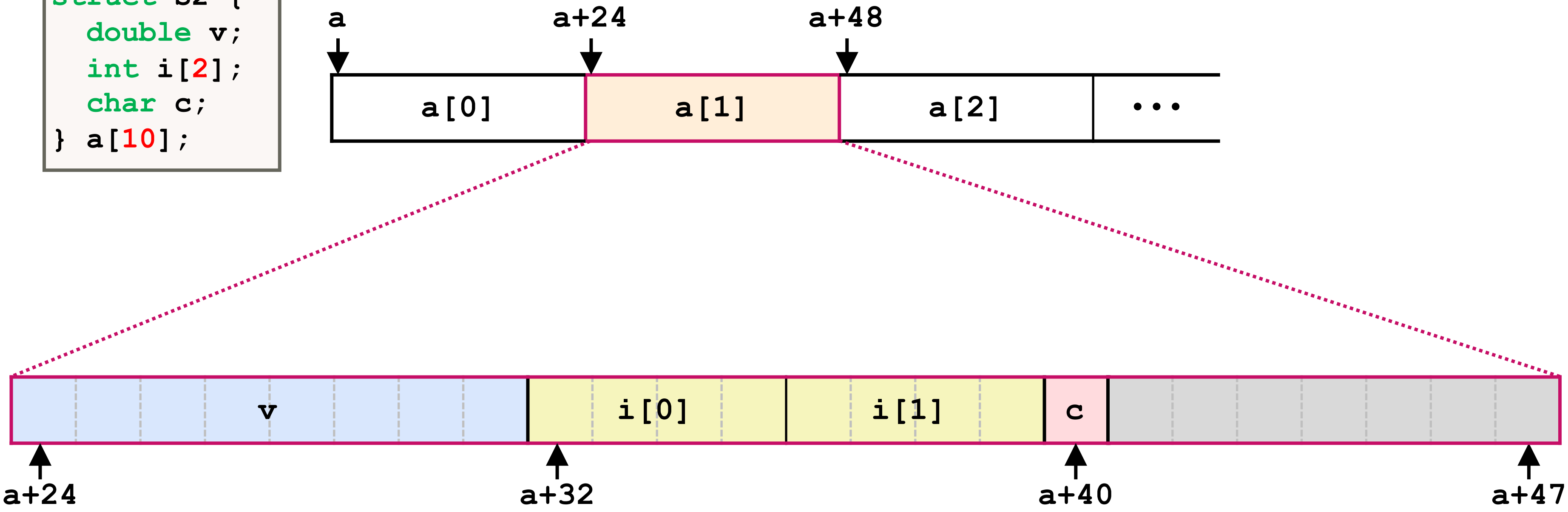
```c
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Arrays of Structures

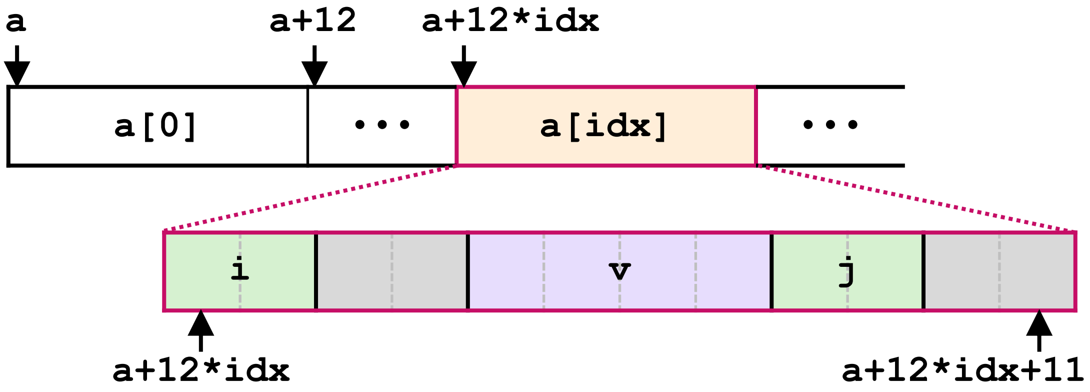
- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int idx){
  return a[idx].j;
}
```

```
# %rdi = idx
leaq    (%rdi,%rdi,2),%rax  # 3*idx
movzwl  a+8(,%rax,4),%eax
ret
```

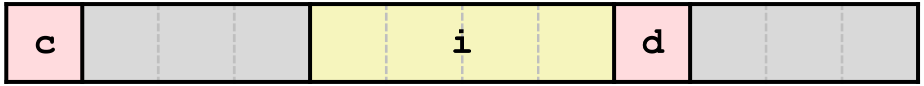
- Compute array offset **12*idx**
  - **sizeof(S3)=12**, including alignment spacers
- Element **j** is at offset 8 within structure
- Assembler gives offset **a+8**, which is resolved during linking

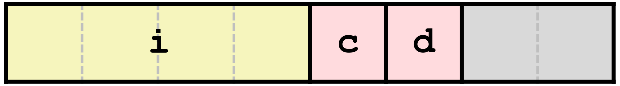
Saving Space

- Put large data types first

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

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





Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct{
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how foo would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding

http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf

Example Struct Exam Question (Cont.)

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct{
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

2. Rearrange the elements of foo to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding

http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf

Lecture Agenda

- Arrays
  - One-Dimensional
  - Multi-Dimensional (Nested)
  - Multi-Level

- Structures
  - Allocation
  - Access
  - Alignment

- Floating Point

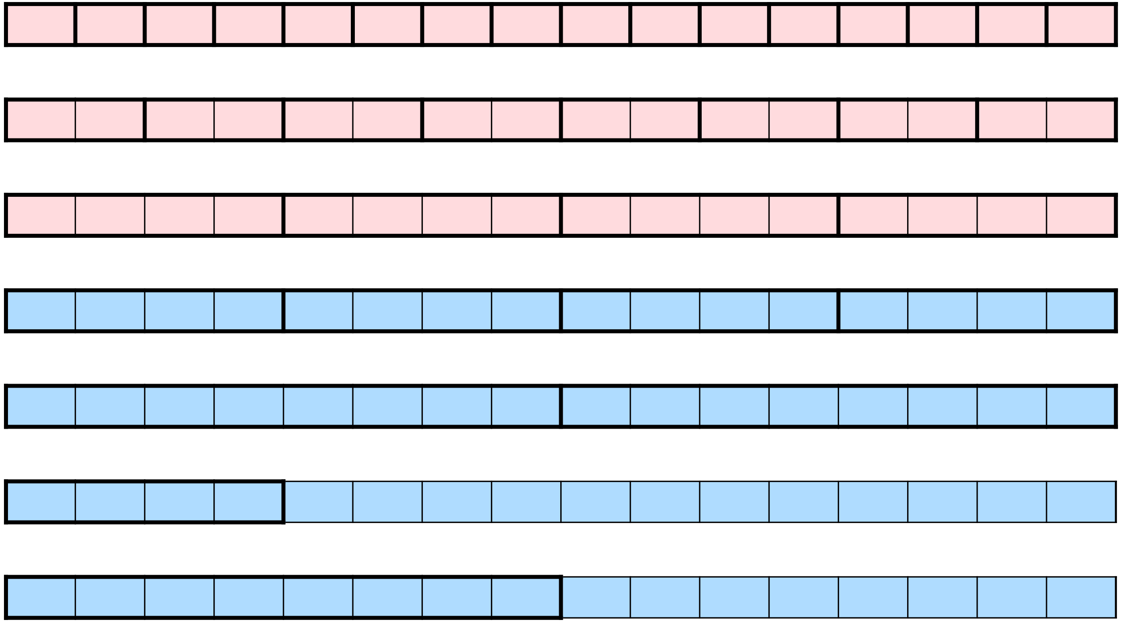
Background

- History
  - x87 FP
    - Legacy, very ugly
  - SSE (Streaming SIMD Extensions) FP
    - SIMD: Single Instruction Multiple Data
    - Supported by old machines
    - Special case use of vector instructions

- AVX (Advanced Vector Extensions) FP
  - Newest version
  - Similar to SSE
  - Documented in book

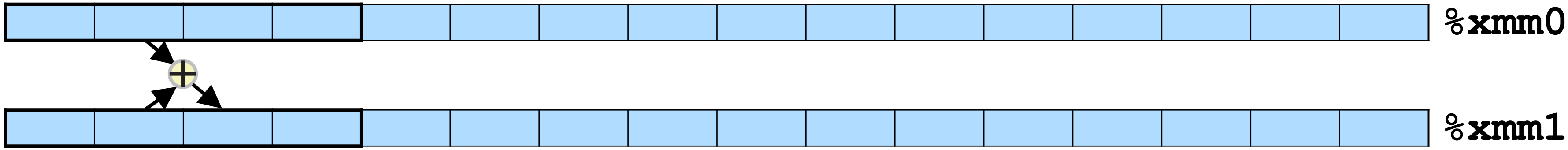
Programming with SSE3

- XMM Registers: 16 total, each 16 bytes
  - 16 single-byte integers
  - 8 16-bit integers
  - 4 32-bit integers
  - 4 single-precision floats
  - 2 double-precision floats
  - 1 single-precision float
  - 1 double-precision float

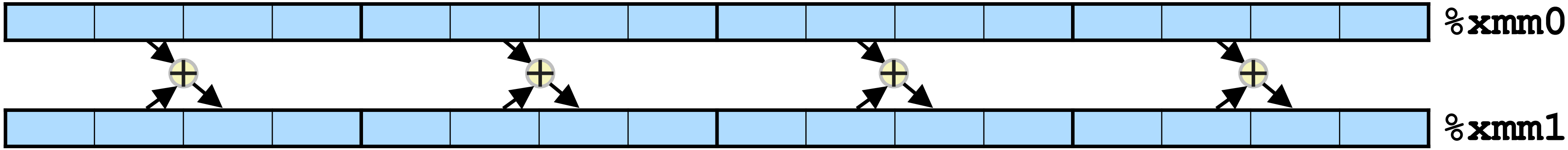


Scalar & SIMD Operations

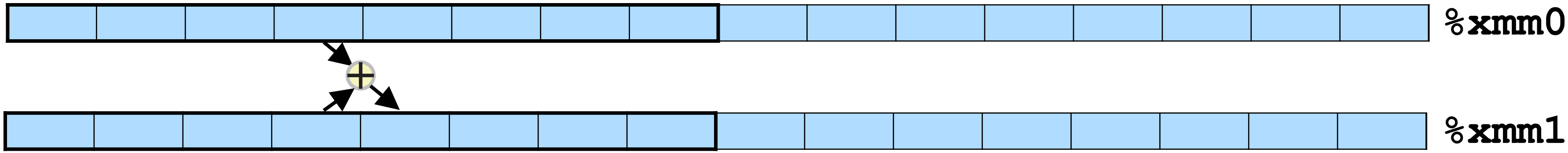
- Scalar operations: single precision `addss %xmm0,%xmm1`



- SIMD operations: single precision `addps %xmm0,%xmm1`



- Scalar operations: double precision `addsd %xmm0,%xmm1`



FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, ...
- Result returned in `%xmm0`
- All XMM registers caller-saved

```c
float fadd(float x, float y){
    return x + y;
}
```

```c
double dadd(double x, double y){
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addss   %xmm1, %xmm0
ret
```

```
# x in %xmm0, y in %xmm1
addsd   %xmm1, %xmm0
ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different **`mov`** instructions to move between XMM registers from the ones to move between memory and XMM registers

```
double dincr(double *p, double v){
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1    # Copy v
movsd   (%rdi), %xmm0   # x = *p
addsd   %xmm0, %xmm1    # t = x + v
movsd   %xmm1, (%rdi)   # *p = t
ret
```

Other Aspects of FP Code

- Lots of instructions
  - Different operations, different formats, ...
- Floating-point comparisons
  - Instructions **`ucomiss`** and **`ucomisd`**
  - Set condition codes **`ZF`**, **`PF`**, and **`CF`**
  - Zeros **`OF`** and **`SF`**

Parity Flag

UNORDERED:	{ZF,PF,CF} ← 111
GREATER_THAN:	{ZF,PF,CF} ← 000
LESS_THAN:	{ZF,PF,CF} ← 001
EQUAL:	{ZF,PF,CF} ← 100

- Using constant values
  - Set %**`xmm0`** register to `0` with instruction **`xorpd %xmm0, %xmm0`**
  - Others loaded from memory

Summary

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment
- Combinations
  - Can nest structure and array code arbitrarily
- Floating point
  - Data held and operated on in XMM registers

[CSED211] Introduction to Computer Software Systems

Lecture 7: Data

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.10.11