Structures, Alignment, Unions

Topics

- Structs in C and assembly layout
- Alignment
- Unions

Announcements

- Bomb Lab #2 update – extension to Friday March 3 by 11:55 pm
  - We are planning to release an alternate bomb server
  - Email any issues to the TAs and Prof. Han
- Read Chapter 3.1-3.12 (except 3.11) and do practice problems

Announcements

- **Midterm #1 Tuesday Feb 28**
  - In class
  - Covers Chapters 1, 2.1-2.3 (no floating point), 3.1-3.9 (no buffer overflow or floating point)
  - Write your answers on the printed exam, then upload them in last 15 minutes to the moodle – bring your laptop!
    - Also turn in paper copy as backup
    - No electronics except during the last 15 minutes of exam
    - You may not use any other online resource except the moodle during the upload time.
    - The moodle will cut off exam submissions at the end of class, so make sure your answers are entered before the end of class!
    - A password to access the midterm on the moodle will be provided in class

Announcements

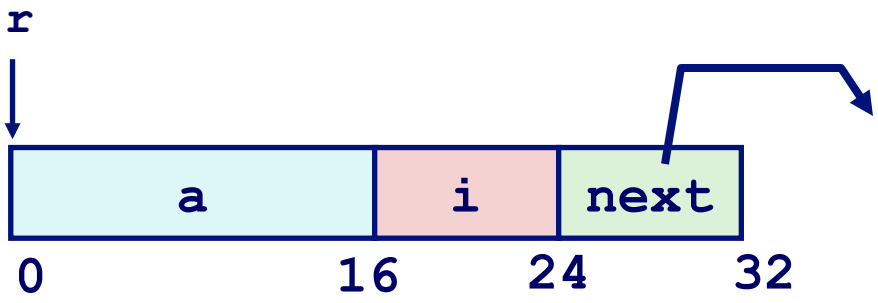
- **Midterm #1 Tuesday Feb 28**
  - Closed book but can bring 1 page summary sheet front & back – write anything you want on it
  - We'll provide a packet of tables from Chapter 3 – print these out and bring them with you
  - TAs may hold extra office hours next Monday
  - We will set up a separate room for those who need extended time for the midterm – contact your TA or the professor to determine when and where to show up (different location than the main lecture hall)
  - Make sure your laptop is charged and functioning properly

Announcements

- **Midterm #1 Tuesday Feb 28**
  - We'll release a practice exam from a previous year later this week - note this was on 32-bit stack conventions, not 64-bit
  - Kinds of questions:
    - Bit manipulation
    - Addition and overflow
    - Call stack
    - C-assembly fill-in-the-blank
  - Good study material:
    - Textbook and its practice problems
    - Lecture slides
    - Practice exam
    - Quizzes

Structure Representation

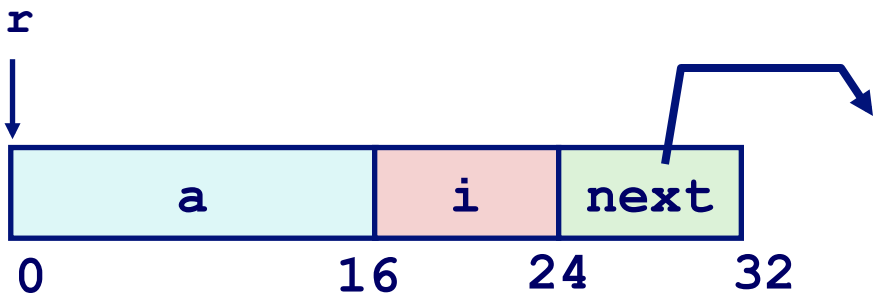
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
} r;
```



- 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

Structure Representation (2)

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



- Refer to members within structure by names
- Members may be of different types

- **Accessing/setting a value of a structure member**

  r.i = 7;

  r.a[0] = 21;

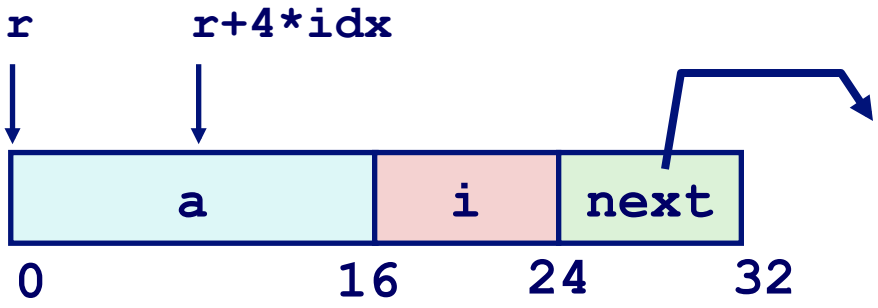
  Suppose q is declared as a "struct rec *q", then

  r.next = q;

  q->i = 4; // equivalent to (*q).i = 4

Generating Pointer to Structure Member

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



- **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];
}
```

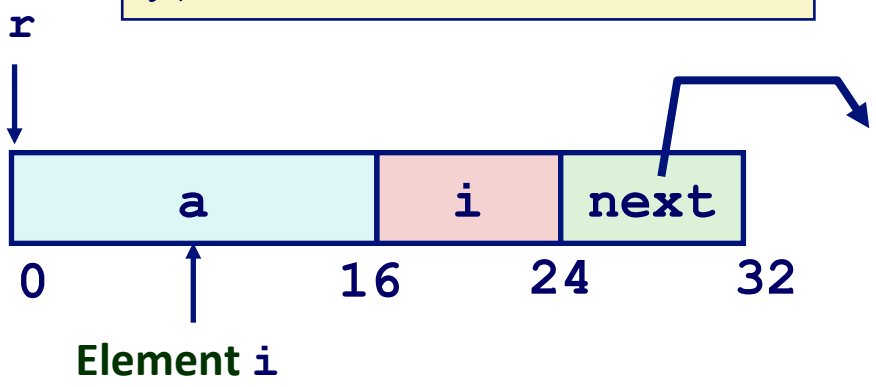
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List

- C Code

```
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;
  }
}
```



Register	Value
%rdi	r
%rsi	val

```
.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
```

Alignment

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64
  - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (double or quad-words) – system-dependent
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans 2 pages

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (x86-64)

Size of Primitive Data Type:

- 1 byte (e.g., `char`)
  - no restrictions on address
- 2 bytes (e.g., `short`)
  - lowest 1 bit of address must be 0_2
- 4 bytes (e.g., `int`, `float`)
  - lowest 2 bits of address must be 00_2
- 8 bytes (e.g., `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

Offsets Within Structure

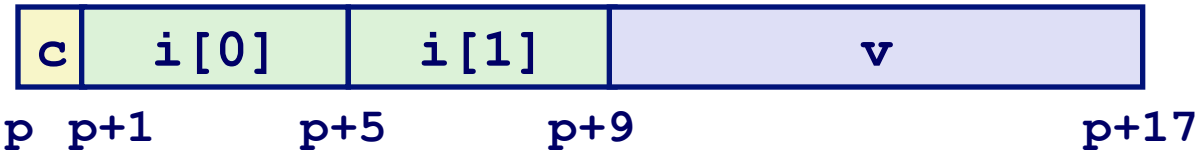
- Must satisfy element's alignment requirement

Overall Structure Placement

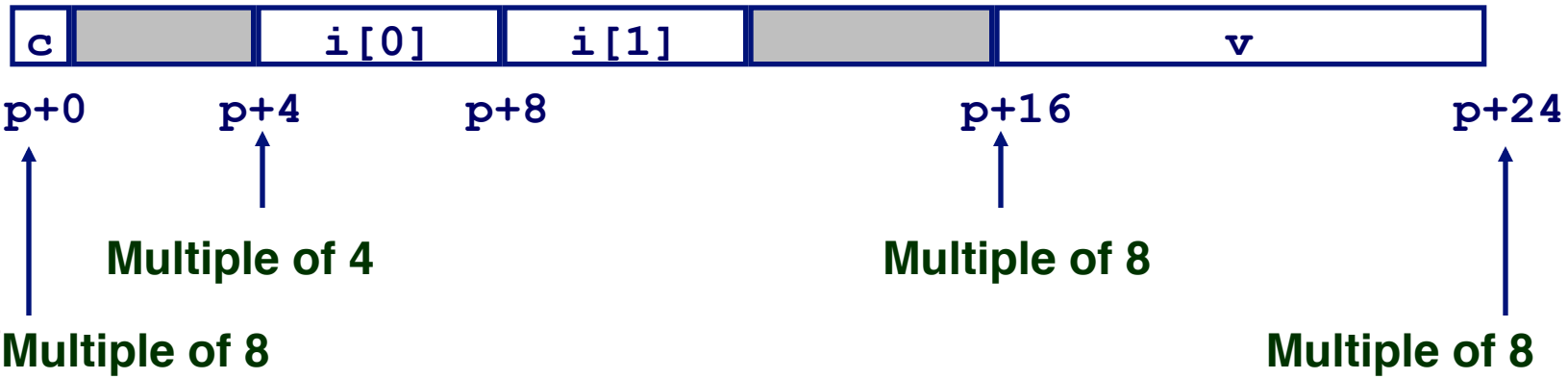
- Each structure has alignment requirement K
  - Largest alignment of any element
- Initial address must be multiple of K
- structure length must be multiple of K

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

Unaligned data:



Aligned data: (K=8 due to `double`)

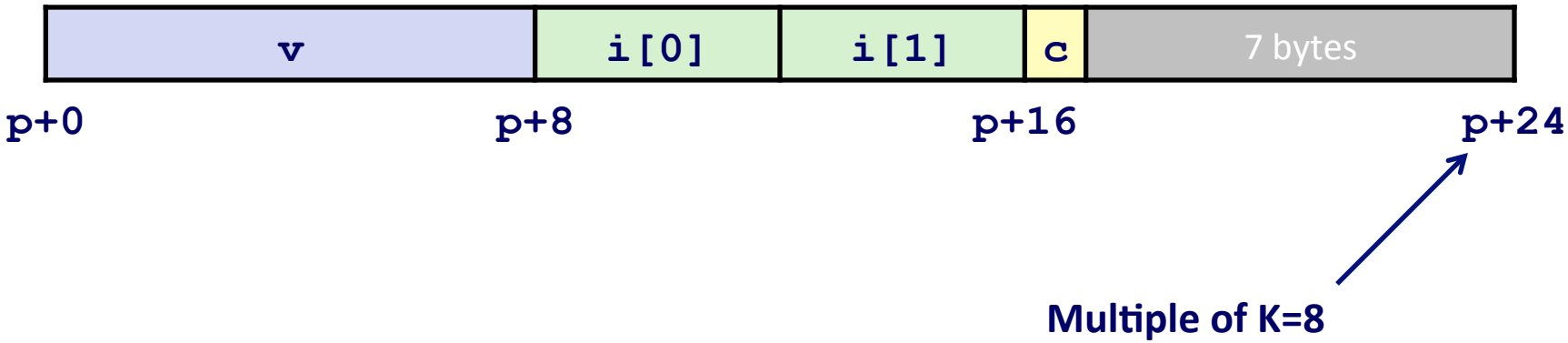


Meeting Overall Alignment Requirement

For largest alignment requirement K

Overall structure must be multiple of K

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

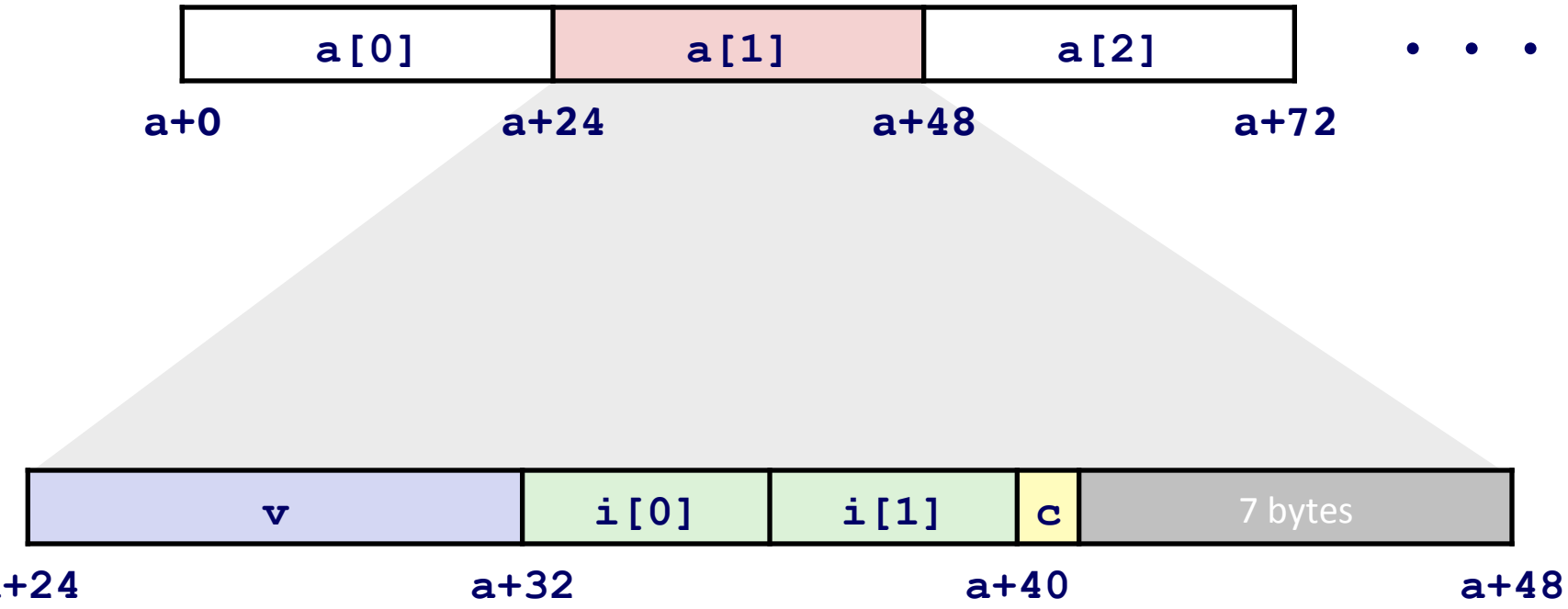


Arrays of Structures

Overall structure length multiple of K

Satisfy alignment requirement for every element

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

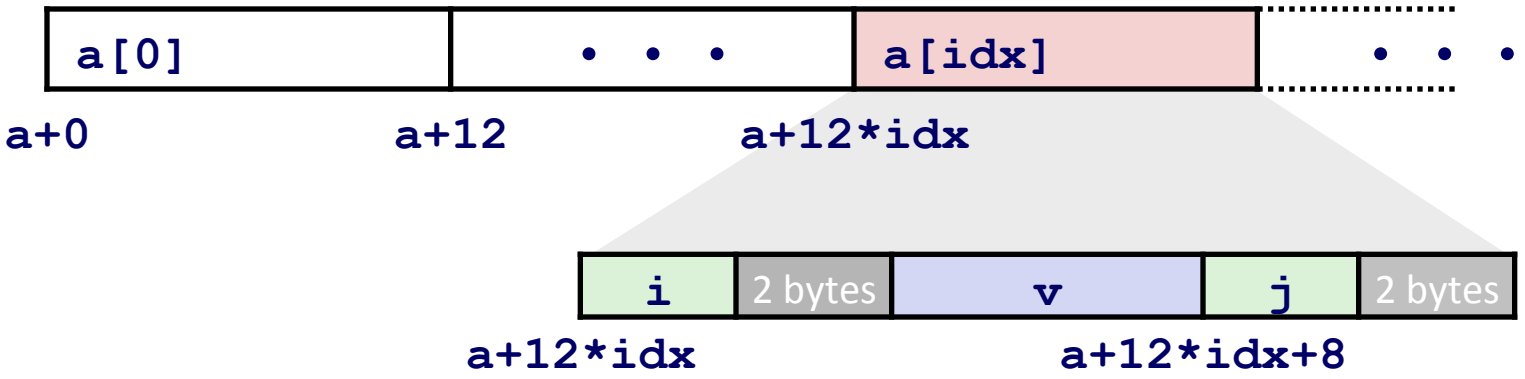


Accessing Array Elements

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

Compute array offset 12*idx

- `sizeof(S3)`, including alignment spacers




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

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

Element `j` is at offset 8 within structure

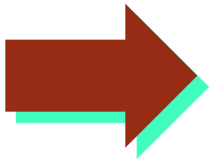
Assembler gives offset `a+8`

- Resolved during linking

Saving Space

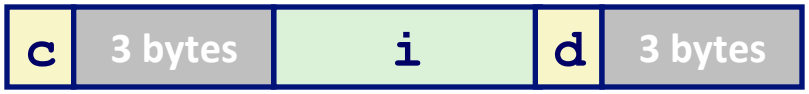
Put large data types first

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



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

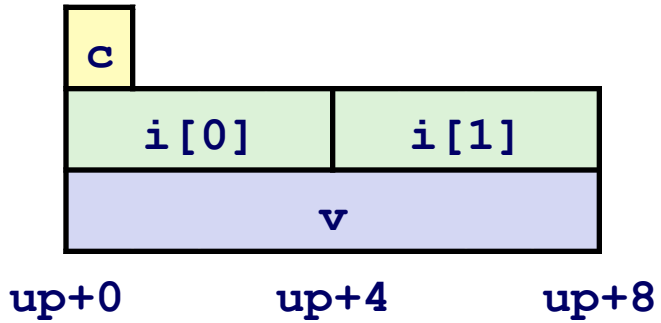
Effect (K=4)



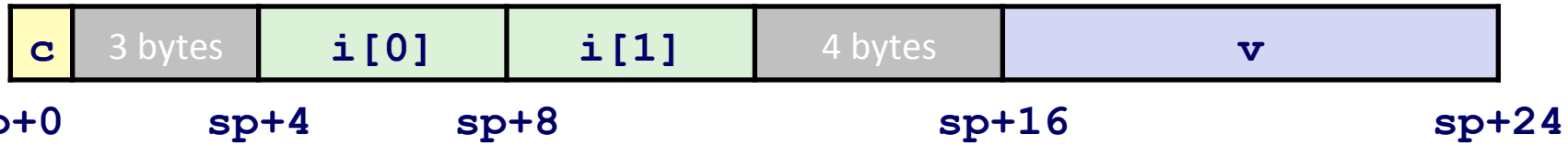
Union Allocation

- Allocate according to largest element – overlay union elements
- Can only use one field at a time

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

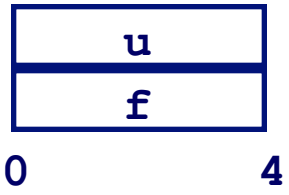



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Unions to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
    converts unsigned int to IEEE fp and rounds, see page 120 of text

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`

Byte Ordering Revisited

Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which byte is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Most significant byte has lowest address
- Sparc

Little Endian

- Least significant byte has lowest address
- Intel x86, ARM Android and IOS

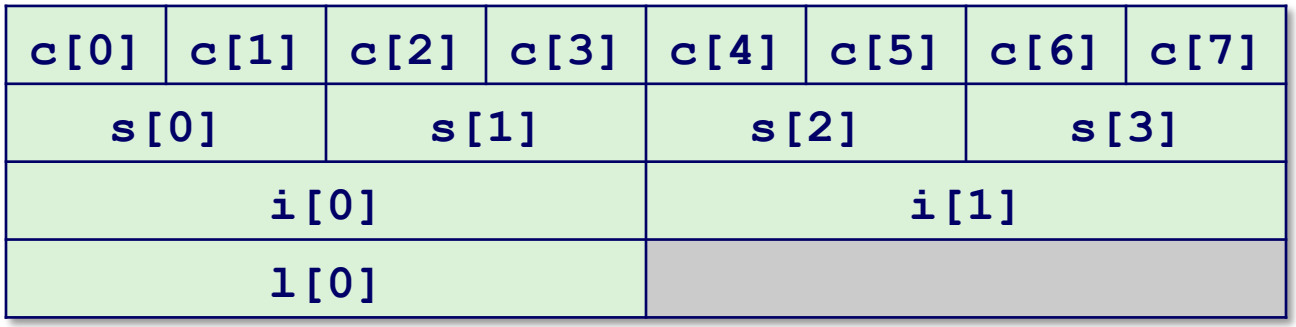
Bi Endian

- Can be configured either way
- ARM

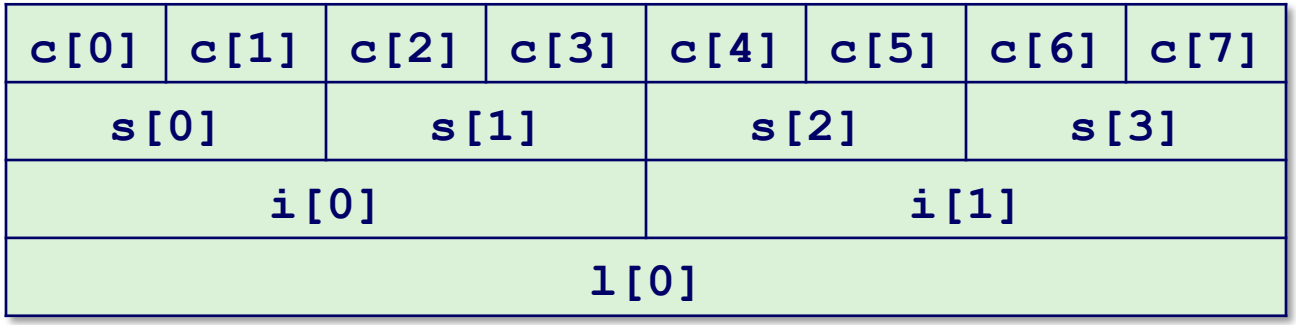
Using Unions to Study Byte Ordering

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit



64-bit

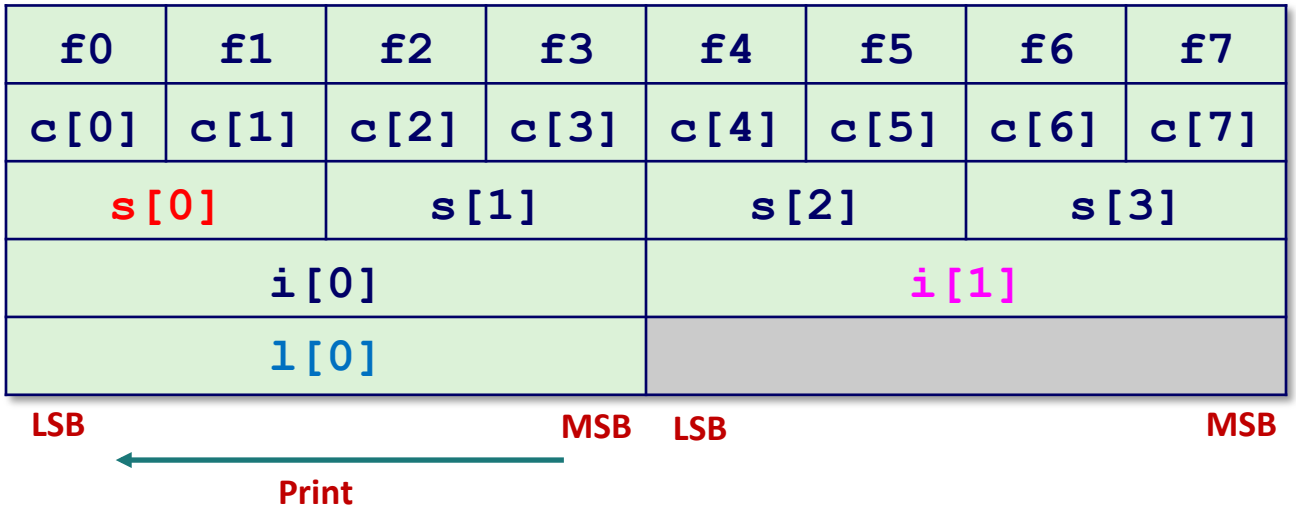


Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==  [0x%x,0x%x,0x%x,0x
%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]
\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

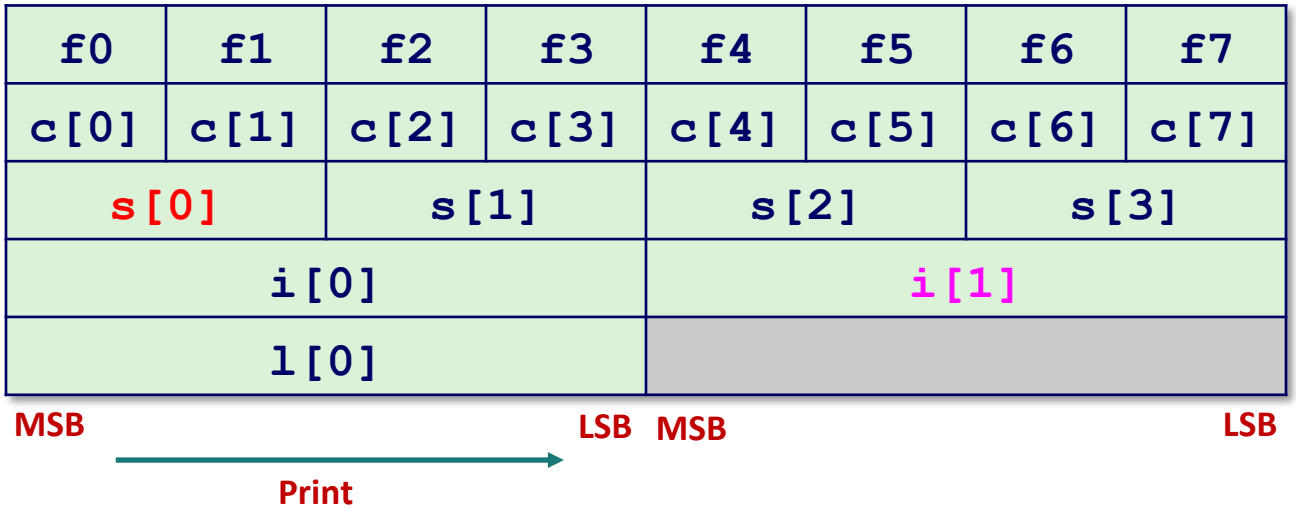


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian

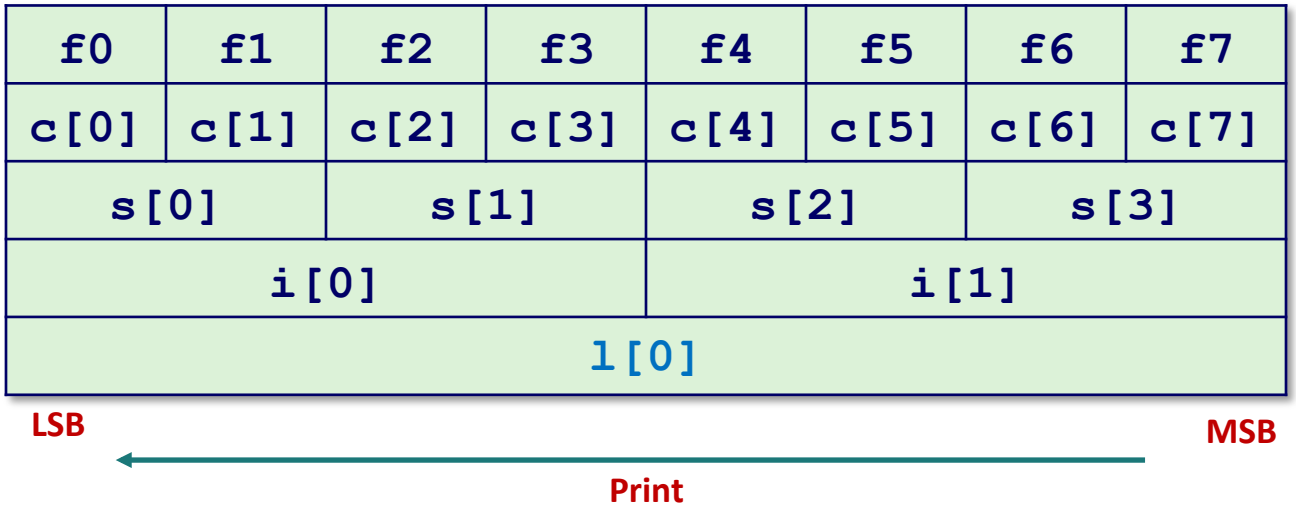


Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Hence the union is useful not just for determining Endianness, but also whether this is a 32-bit or 64-bit machine

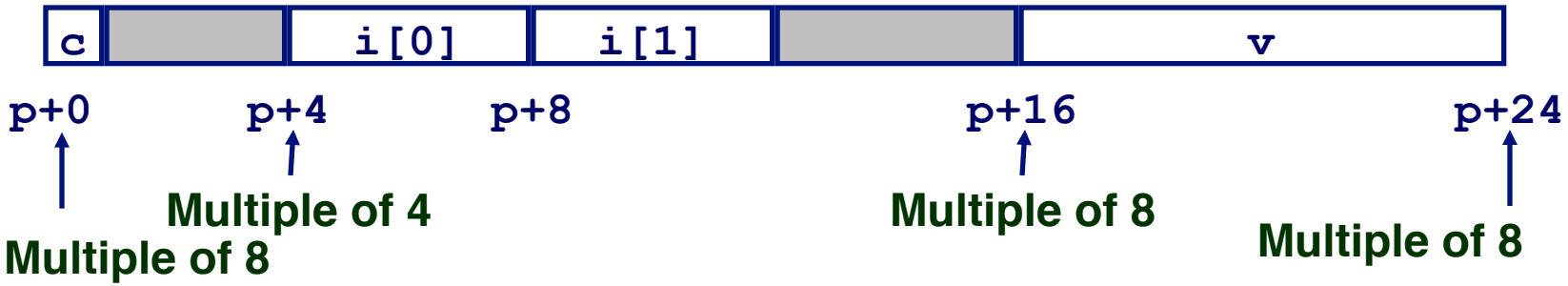
Supplementary Slides

Linux vs. Windows

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

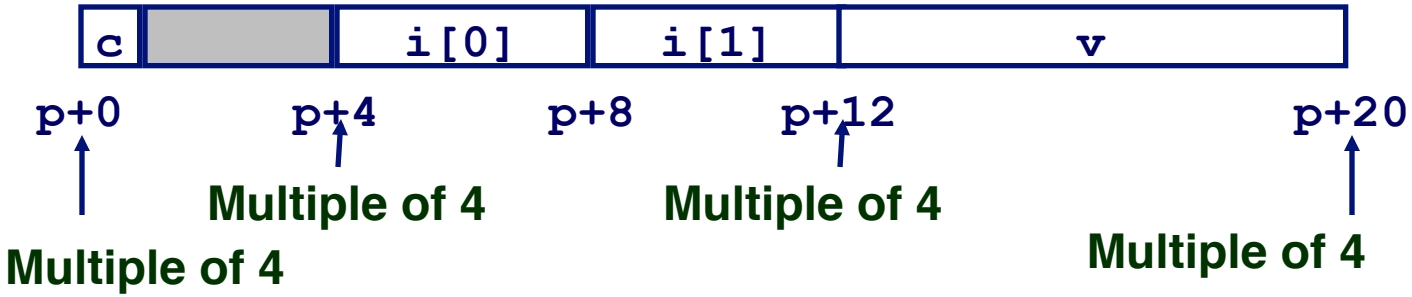
Windows (including Cygwin):

- K = 8, due to `double` element



Linux:

- K = 4; `double` treated like a 4-byte data type



Overall Alignment Requirement

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

p must be multiple of:
- **8 for Windows**
- **4 for Linux**

p+0 p+8 p+12 p+16 **Windows:** p+24 **Linux:** p+20

```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

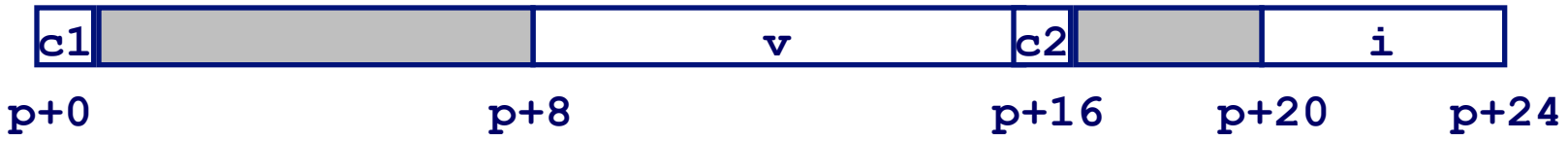
p must be multiple of 4 (in either OS)

p+0 p+4 p+8 p+12 p+16 p+20

Ordering Elements Within Structure

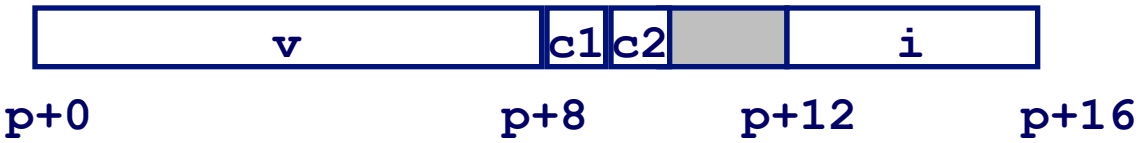
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows




```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

2 bytes wasted space



Basic Data Types

Integral

- Stored & 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`

Floating Point

- Stored & operated on in floating point registers

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

C pointer declarations

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Understanding Pointers & Arrays #1

Decl	*An*			**An*		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						

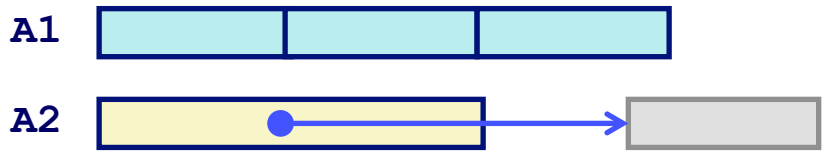
Cmp: Compiles (Y/N)

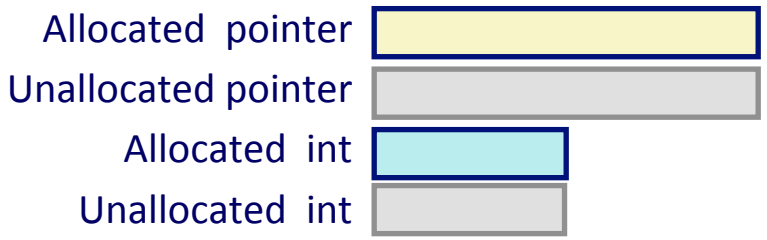
Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

Understanding Pointers & Arrays #1

Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4





Cmp: Compiles (Y/N)

Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

Understanding Pointers & Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									
`int (*A4[3])`									

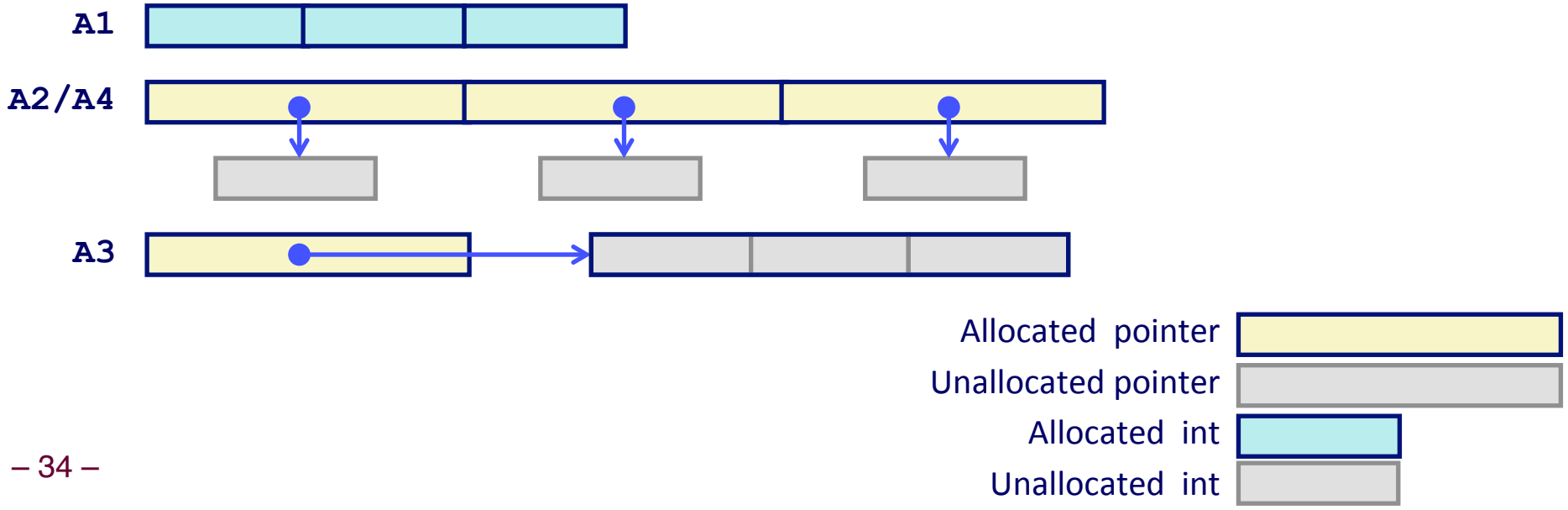
Cmp: Compiles (Y/N)

Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	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
int (*A4[3])	Y	N	24	Y	N	8	Y	Y	4



Understanding Pointers & Arrays #3

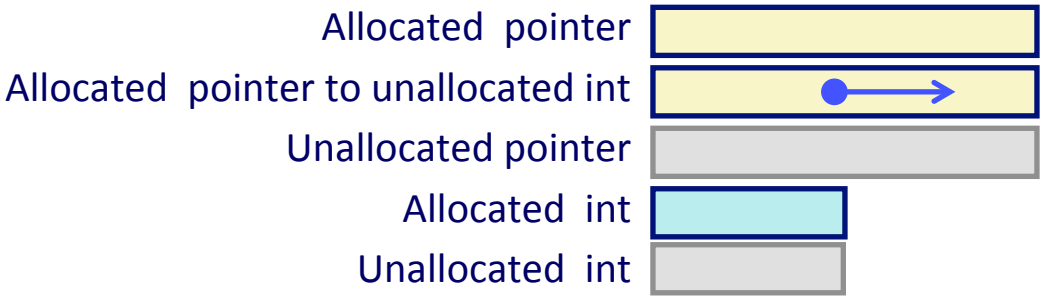
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

Cmp: Compiles (Y/N)

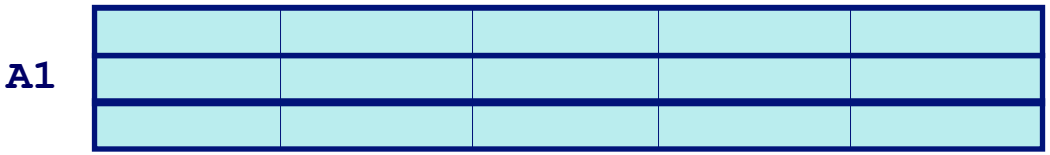
Bad: Possible bad pointer reference (Y/N)

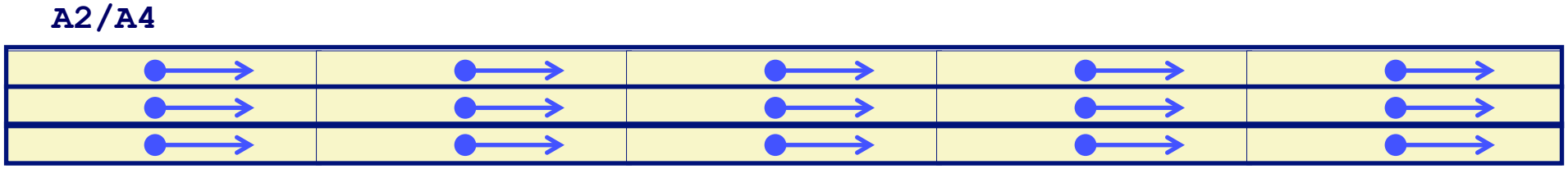
Size: Value returned by `sizeof`

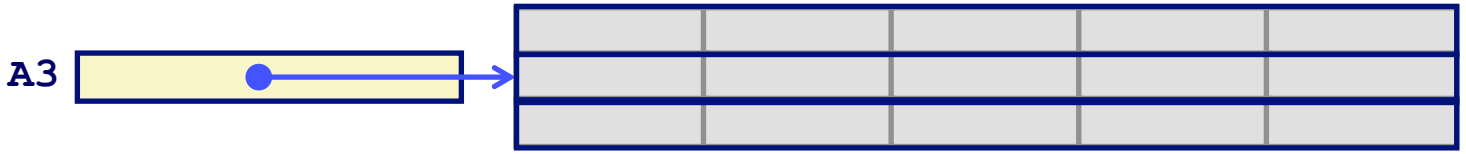
Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

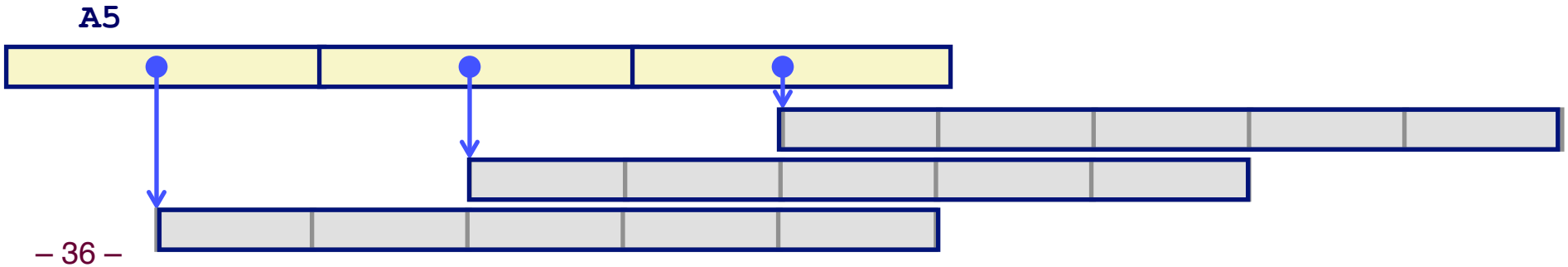


Declaration
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`









Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

Cmp: Compiles (Y/N)

Bad: Possible bad pointer reference (Y/N)

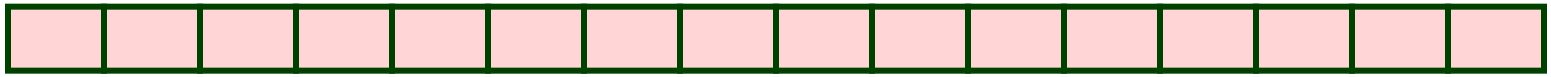
Size: Value returned by `sizeof`

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`	N	–	–
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4

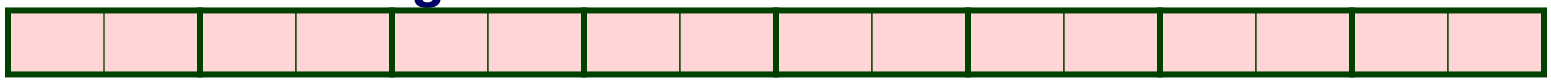
Aside: Programming with SSE3

XMM Registers

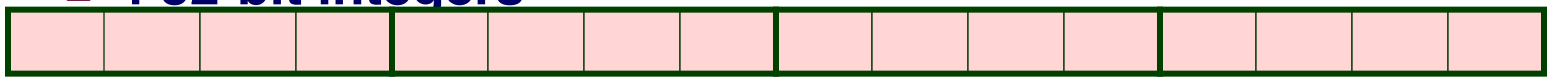
- 16 total, each 16 bytes
- 16 single-byte integers



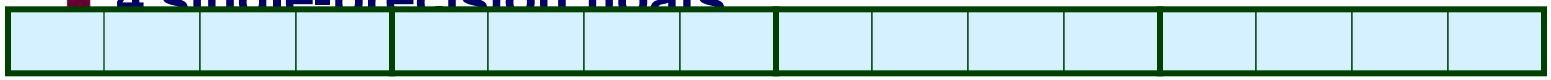
- 8 16-bit integers



- 4 32-bit integers

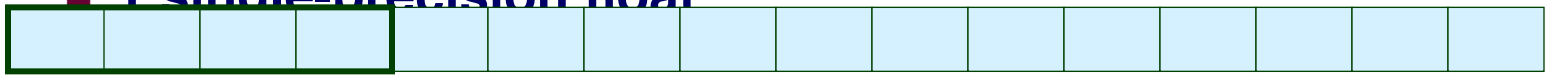


- 4 single-precision floats

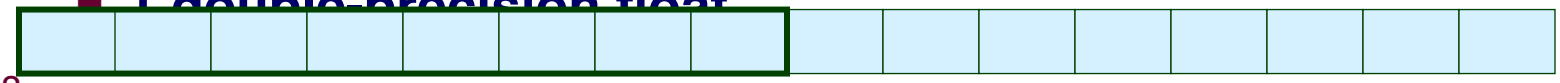


- 2 double-precision floats

- 1 single-precision float

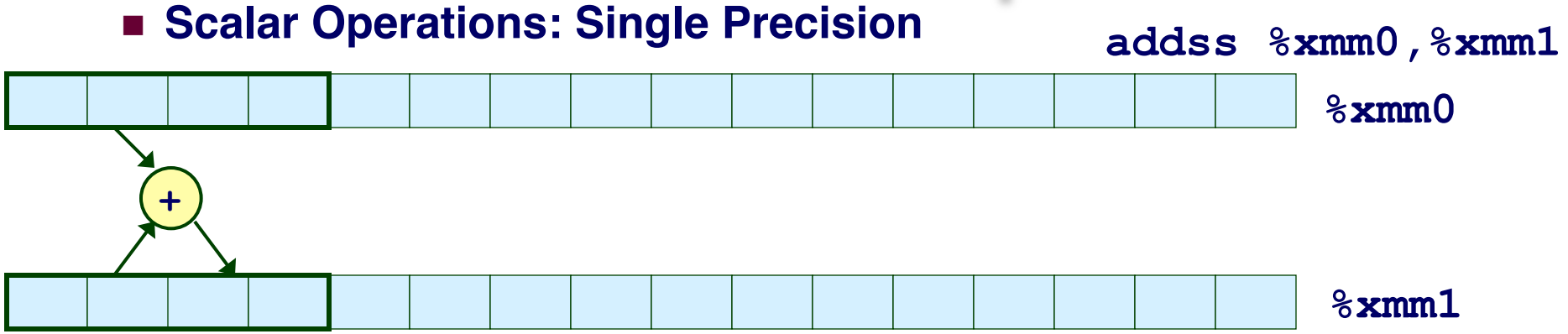


- 1 double-precision float

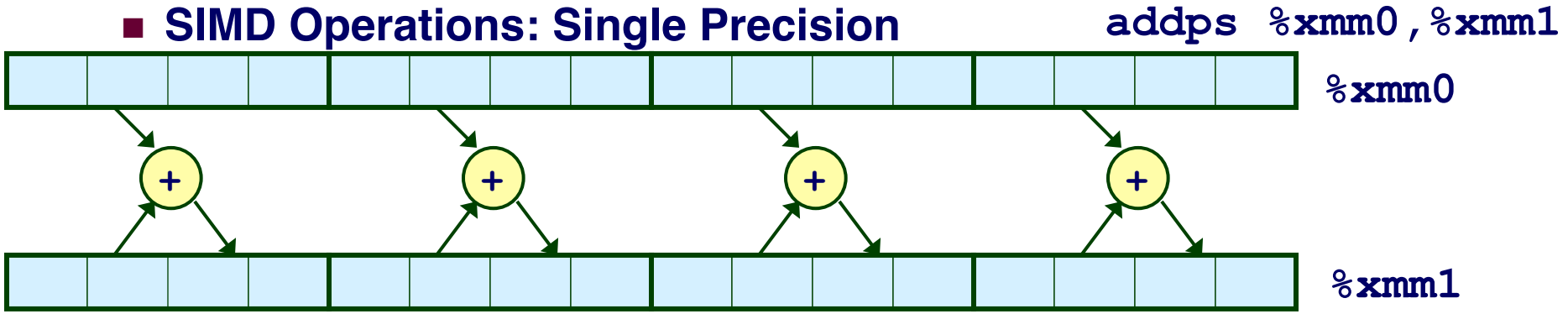


Aside: Scalar & SIMD Operations

- Scalar Operations: Single Precision



- SIMD Operations: Single Precision



- Scalar Operations: Double Precision