CSCI2467: Systems Programming Concepts

Slideset 13: Machine Level IV: Data structures
Source: CS:APP Sections 3.8-3.9, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

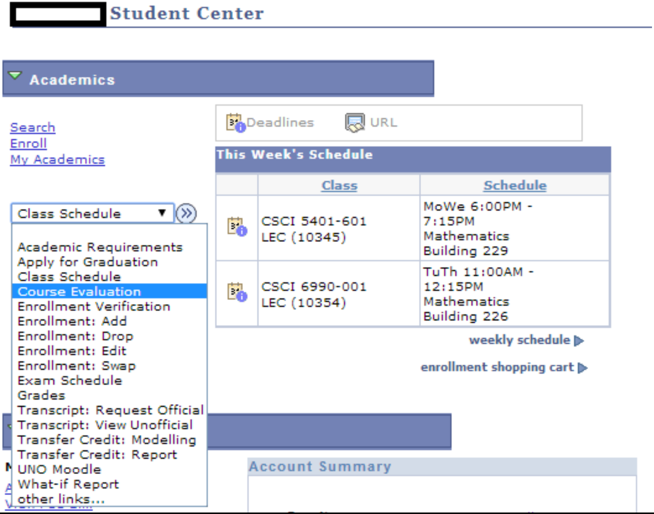
Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY of NEW ORLEANS

Course evaluations: available now

- Please evaluate course on WebStar!
  - Both positive and negative feedback welcome (and confidential)



Course evaluations: available now

CSCI-5401-601 | Principles Operating Systems I | 2015 Fall | Regular Academic Session

1. The instructor demonstrated an enthusiasm for teaching this course.

   Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree | Cannot Assess

2. The instructor stimulated my interest.

   Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree | Cannot Assess

3. The instructor was prepared and well organized.

   Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree | Cannot Assess

4. The instructor's manner of communicating was easy to understand.

   Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree | Cannot Assess

5. The instructor's lectures, explanations, and feedback were clearly presented.

   Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree | Cannot Assess

attacklab!

Lab notes

- using `gdb` with input redirection
- how to use `hex2raw` to generate inputs
- practice monitoring the stack with `gdb`:

(gdb) `print $rsp` and `x $rsp`

(gdb) `info frame`

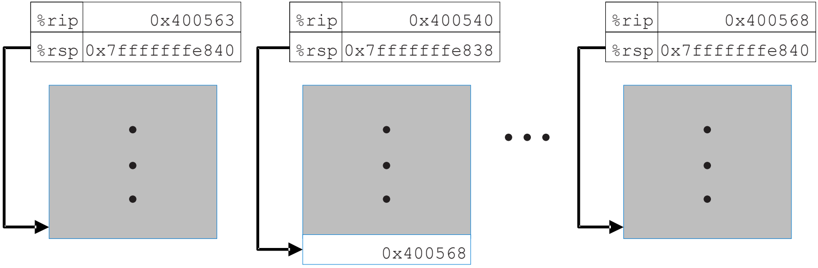
Don't forget: due Monday (April 20 11:59pm)

attacklab – more notes

- memory addresses in buffer overflow will be *little-endian*
  - gdb can help you sort out endianness:

    see differences between `x/4b $rsp` and `x/1w $rsp`

    (also "giant" 64-bit words: `x/1gx $rsp`)
- refer to Appendix A for reminders on `hex2raw` usage



(a) Executing `call` (b) After `call` (c) After `ret`

Today

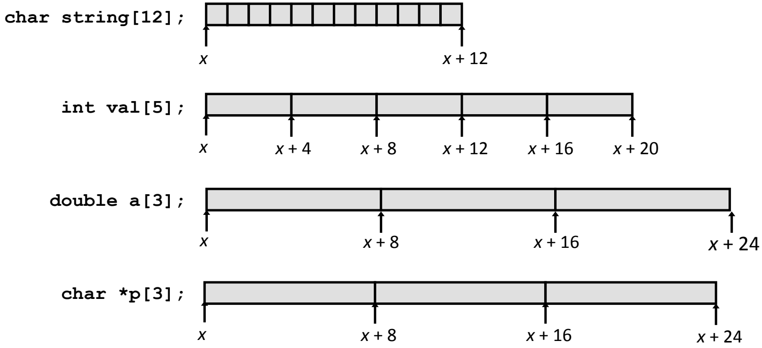
- Course notes
1. Arrays
   - One-dimensional
   - Multi-dimensional
   - Multi-level
2. Structures
   - Allocation
   - Access
   - Alignment
3. Test our knowledge

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



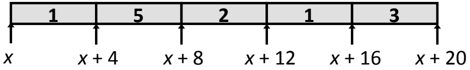
Array access

Basic Principle

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

- Array of data type *T* and length *L*
- Identifier **A** can be used as a pointer to array element 0: Type *T**

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

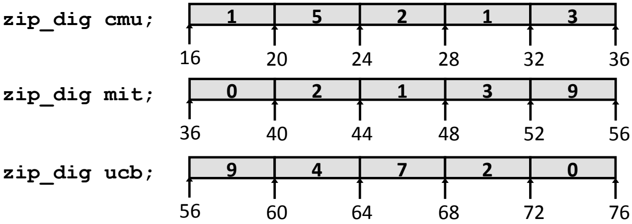


Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	$x + 4$
&val[2]	int *	$x + 8$
val[5]	int	??
*(val+1)	int	5
val + *i*	int *	$x + 4i$

Array example

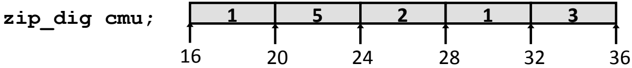
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



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

Array access example




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

- **Register `%rdi` contains starting address of array**

```
# rdi = z
# rsi = digit (array index)
mov     eax, DWORD PTR [rdi+rsi*4]
```

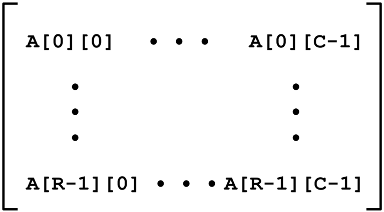
Array loop example

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

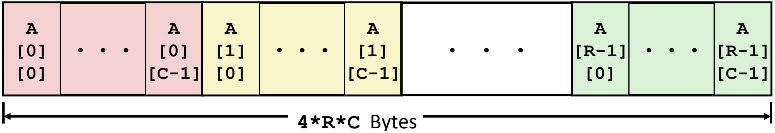
```
# rdi = z
     mov     eax, 0               #   i = 0
     jmp     .L2                  #   goto L2
.L3:
     add   DWORD PTR [rdi+rax*4], 1 # z[i]++
     add     rax, 1               # i++
.L2:
     cmp     rax, 4               # i:4
     jbe     .L3                  # if <= goto L3
     rep ret
```

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**
  - Row-Major Ordering

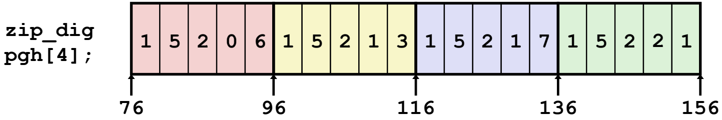


`int A[R][C];`



Nested array example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

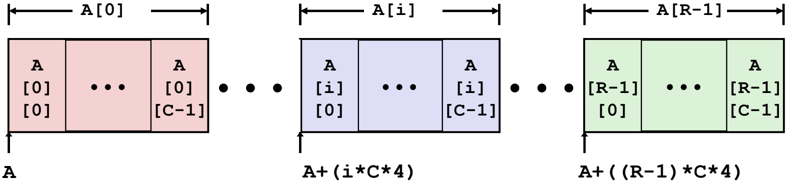


- “**`zip_dig pgh[4]`**” **equivalent to “`int pgh[4][5]`”**
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**’s, allocated contiguously
- **“Row-Major” ordering of all elements in memory**

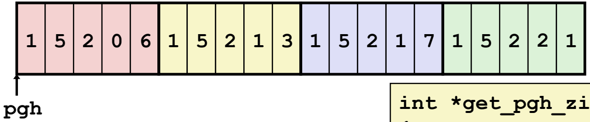
Nested array row access

- **Row Vectors**
  - `A[i]` is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address `A +` *i* * (*C* * *K*)

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



Nested array row access code




```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
# rdi = index
lea     rdx, [rdi+rdi*4]
lea     rax, [0+rdx*4]
add     rax, OFFSET FLAT:pgh
```

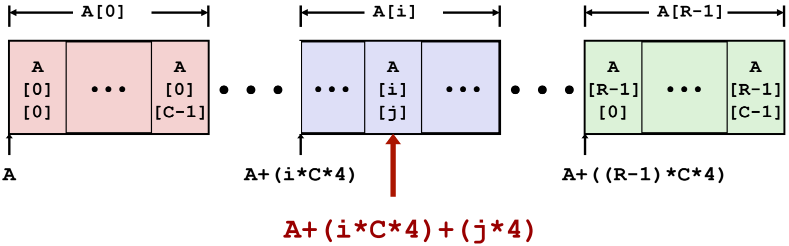
Row vector:

- `pgh[index]` is array of 5 `ints`
  - starting address: `pgh + (20 * index)`
- Machine code: computes and returns address
  `= pgh + 4*(index + 4*index)`

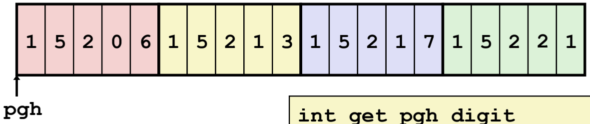
Nested array element access

- **Array Elements**
  - **`A[i][j]`** is element of type *T*, which requires *K* bytes
  - Address **`A + `** $i * (C * K) + j * K = A + (i * C + j) * K$

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



Nested array element access code




```c
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```

```
lea      rax, [rdi+rdi*4]
add      rsi, rax
mov      eax, DWORD PTR pgh[0+rsi*4]
```

Array Elements:

- pgh[index][dig] is int
- address: pgh + (20 * index) + (4 * dig)

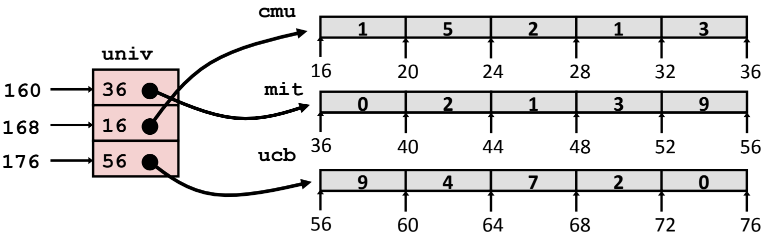
  = pgh + 4*(5*index + dig)

Multi-level array example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

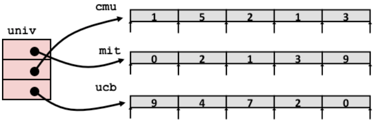
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- **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 a multi-level array

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




```
    sal       rsi, 2
    add       rsi, QWORD PTR univ[0+rdi*8]
    mov       eax, DWORD PTR [rsi]
    ret
```

- Must do two memory reads:
- first get pointer to row array
- then access element within array

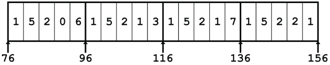
Array element access

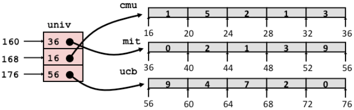
Nested array

```
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]`**

Course notes

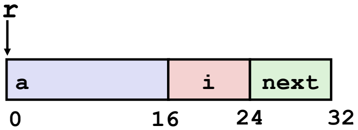
1. Arrays
   - One-dimensional
   - Multi-dimensional
   - Multi-level

2. Structures
   - Allocation
   - Access
   - Alignment

3. Test our knowledge

Structure representation

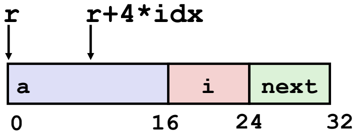
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 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[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`**

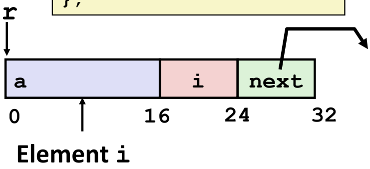
```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

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

Following Linked List

- **C Code**

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

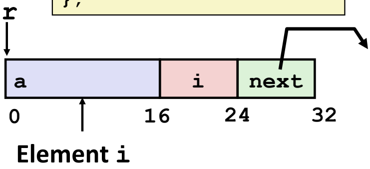
```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

Following Linked List

- **C Code**

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

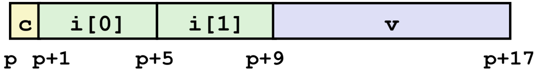


Register	Value
%rdi	r
%rsi	val

```
  jmp    .L2             # test r first
.L3:
  movsx rax, DWORD PTR [rdi+16]      # i = M[r+16]
  mov   DWORD PTR [rdi+rax*4], esi # M[r+4*i] = val
  mov   rdi, QWORD PTR [rdi+24]      # r = M[r+24]
.L2:
  test  rdi, rdi      # test r
  jne   .L3           # if !=0 jump to top of loop
```

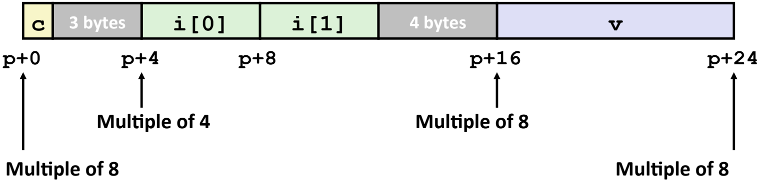
Structures & Alignment

- **Unaligned Data**




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

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



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
    - Virtual memory trickier when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

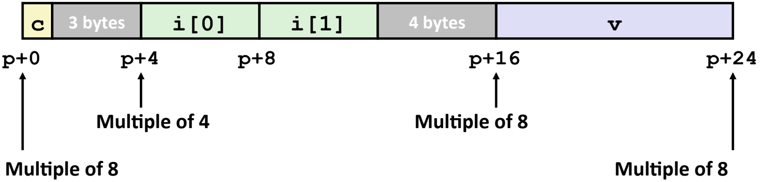
Specific cases of alignment (x86-64)

- **1 byte: `char`, ...**
  - no restrictions on address
- **2 bytes: `short`, ...**
  - lowest 1 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

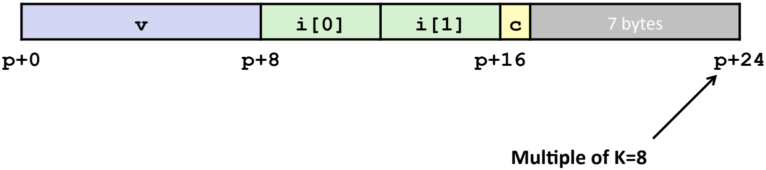
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting overall alignment requirement

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

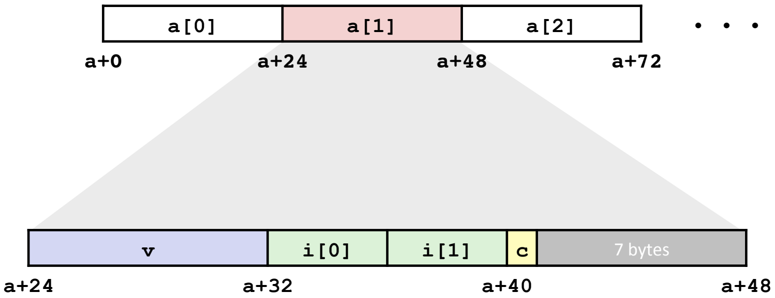
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of structures

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

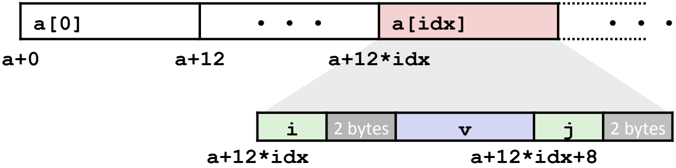
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

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




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

```
lea    rax, [rdi+rdi*2]#rdi*3
sal    rax, 2       # 4*(rdi*3)
movzx  eax, WORD PTR a[rax+8]
```

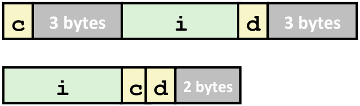
Saving space

- **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)**



Summary

- Arrays
  - elements packed into contuguous region of memory
  - use index arithmetic to locate individual elements
- Structures
  - elements packed into single region of memory
  - access using offsets determined by compiler
  - possibly require internal and external padding to ensure alignment
- Combinations
  - can nest structure and array code arbitrarily

Course notes

1. Arrays
   - One-dimensional
   - Multi-dimensional
   - Multi-level

2. Structures
   - Allocation
   - Access
   - Alignment

3. Test our knowledge

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`**

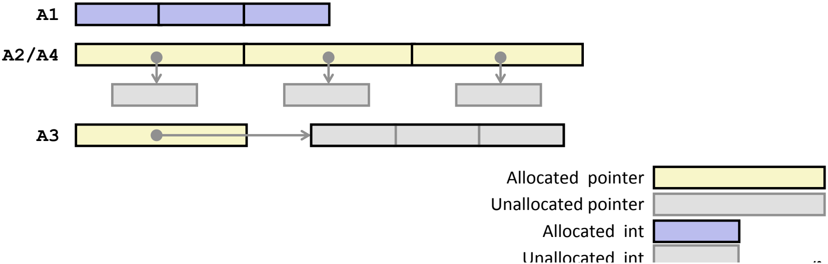
Understanding pointers & arrays #2

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



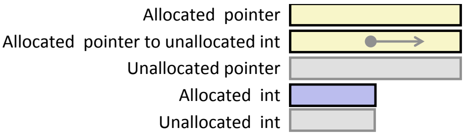
Understanding pointers & arrays #3

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]`									

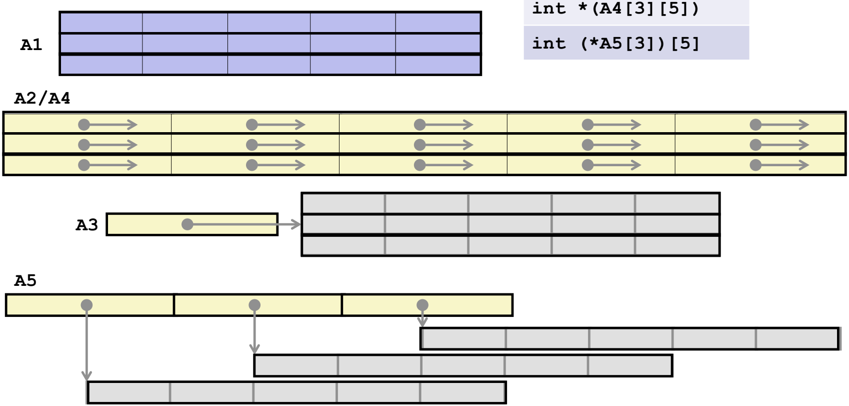
- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

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]`			

Understanding pointers & arrays #3



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