Announcement

- **Short Office Hour today: until 6pm**
- **Lab 1 regrade**
  - if you failed to test your code on linuxlab machines using driver.pl then:
  - TEST your code on linuxlab machine using driver.pl and see if you lose points due to violation of programming rules
  - If you are surprised by this, you can fix your code and push the changes.
  - We will regrade your lab1, and assign you:
    min(old code running with btest,
    the correctness portion of new code running with driver.pl)
    (Your lab1 score will incur some penalty, but it’s better than 0.)
  - NOTE that this won’t help you
    - if you have gotten a decent score already running with driver.pl the first time around; or
    - If you have not implemented the correct functionalities and gotten 0 with btest the first time around

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - **Illustration of Recursion**

Recursive Function

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
`%rdi`	`x`	Argument
`%rax`	Return value	Return value

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

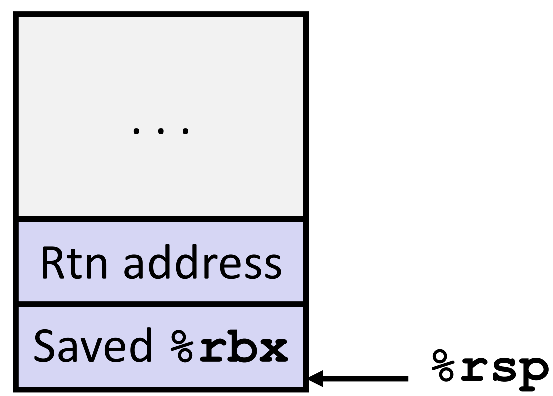
Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
`%rdi`	`x`	Argument

Why? Because %rbx is callee-save.

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   ret
```



Recursive Function Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

. . .

Rtn address

Saved %rbx

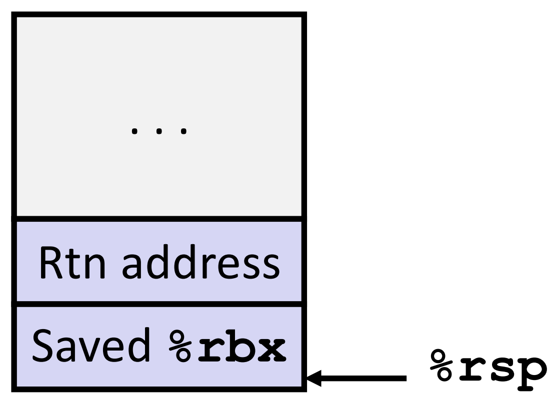
%rsp

Recursive Function Call

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Recursive call return value	



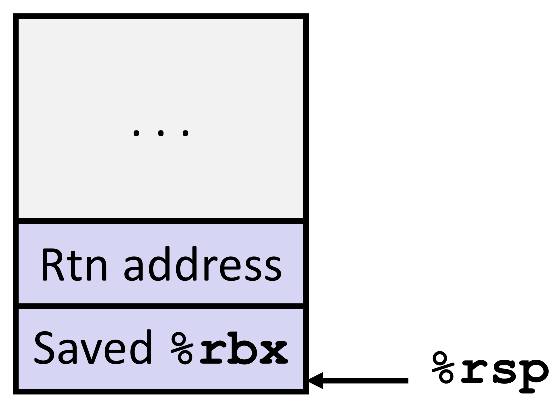
We have our local state in %rbx, and the recursive pcount_r has the right argument.

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

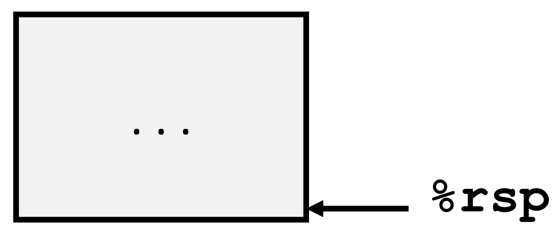


Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
`%rax`	Return value	Return value

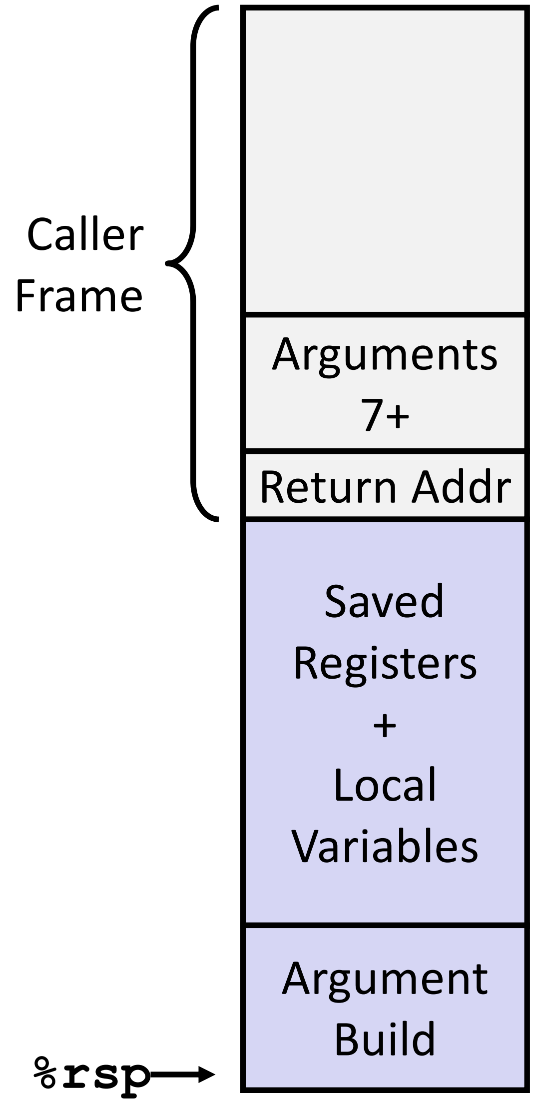


Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P

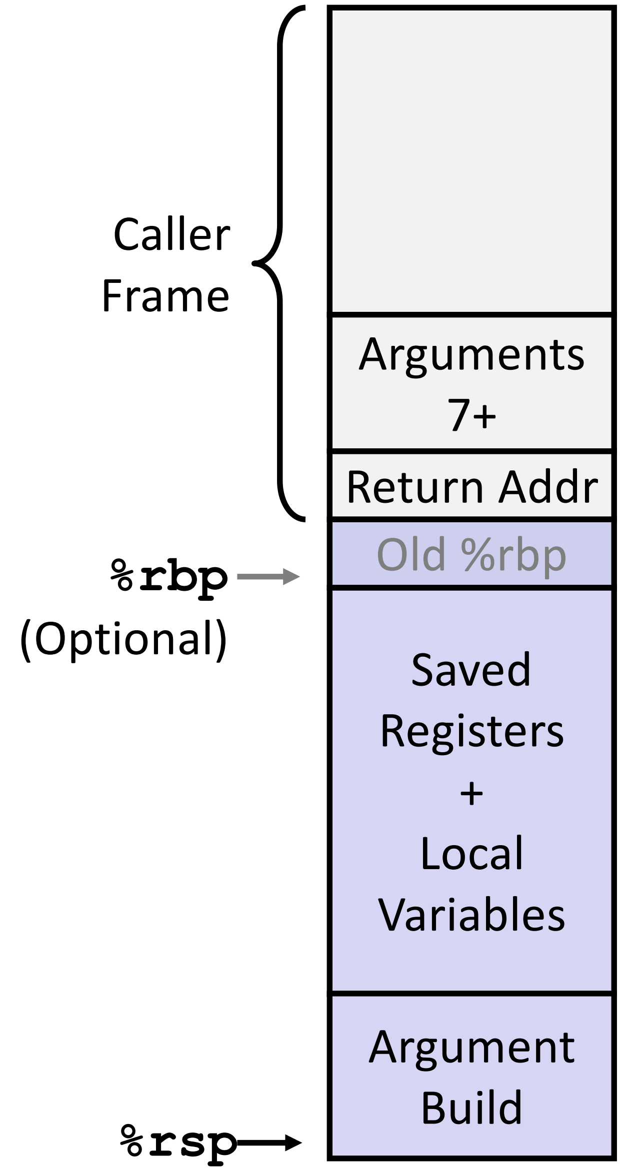
x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`



x86-64 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- `%rbp` may be optionally used as frame pointer when the compiler does not know the stack frame size.



Machine-Level Programming IV: Compound Data Types

B&O Readings: 3.8-3.9

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Today: Compound Types in C

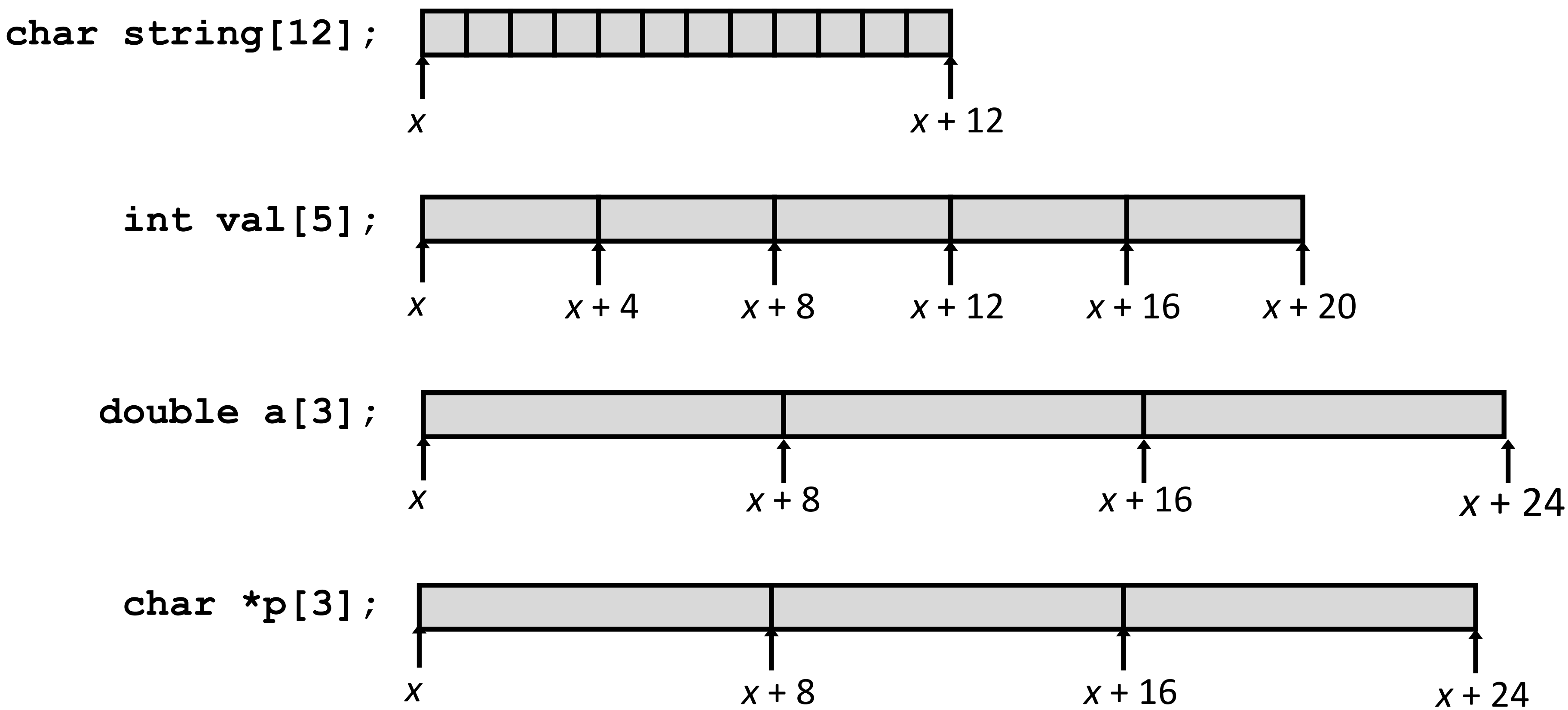
- **Arrays**
  - One-dimensional arrays
  - Multi-dimensional / nested arrays
  - Multi-level arrays
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Unions**

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



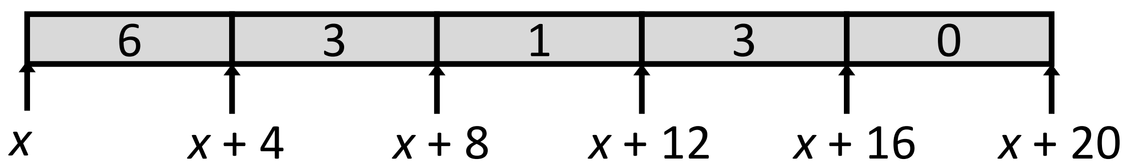
Array Access

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**

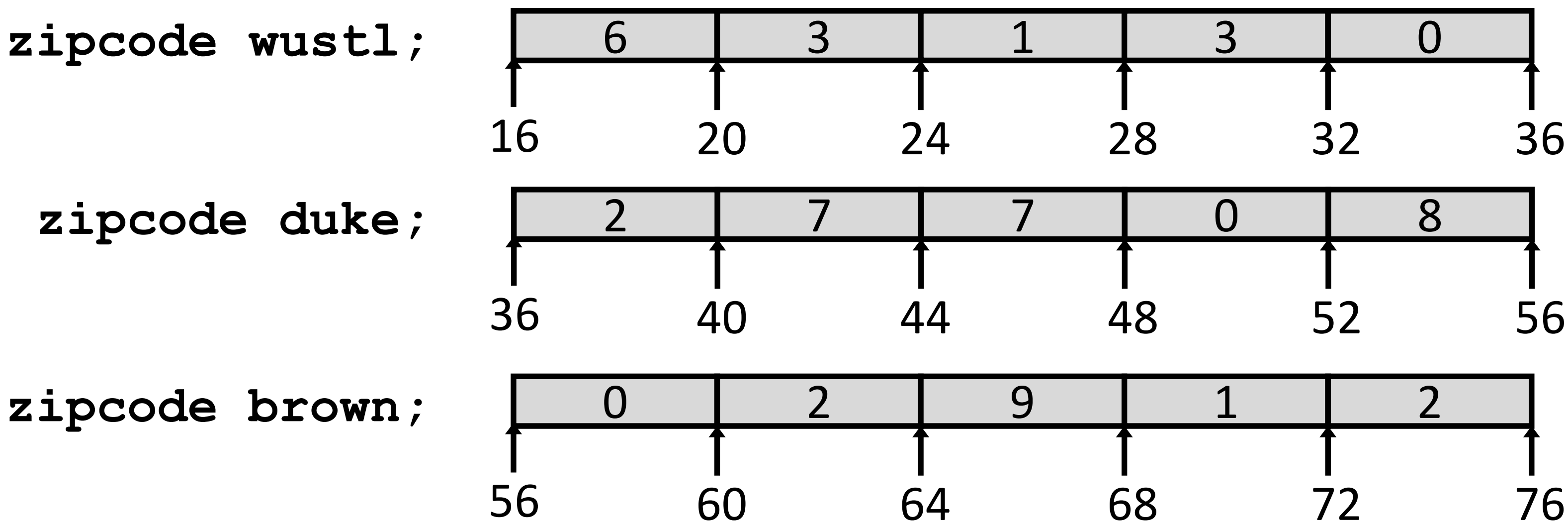
`int val[5];`



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

Array Example

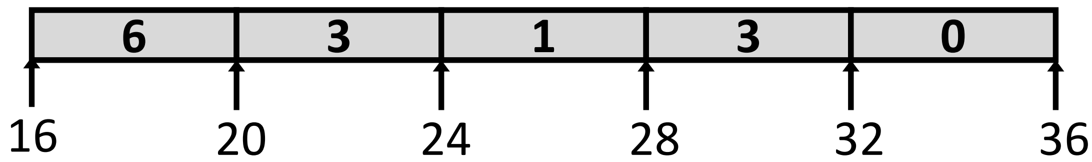
```
#define ZIP_LEN 5
typedef int zipcode[ZIP_LEN];

zipcode wustl = { 6, 3, 1, 3, 0 };
zipcode duke  = { 2, 7, 7, 0, 8 };
zipcode brown = { 0, 2, 9, 1, 2 };
```



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

Array Accessing Example

`zipcode wustl;`




```
int get_digit
  (zipcode 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 `%rdi + %rsi*4`
- Use memory reference `(%rdi,%rsi,4)`

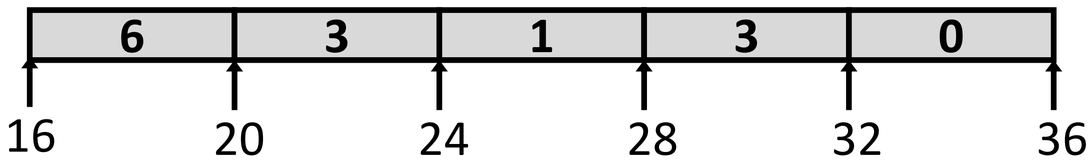
Array Loop Example

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

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

Array Accessing Puzzle

`zipcode wustl;`



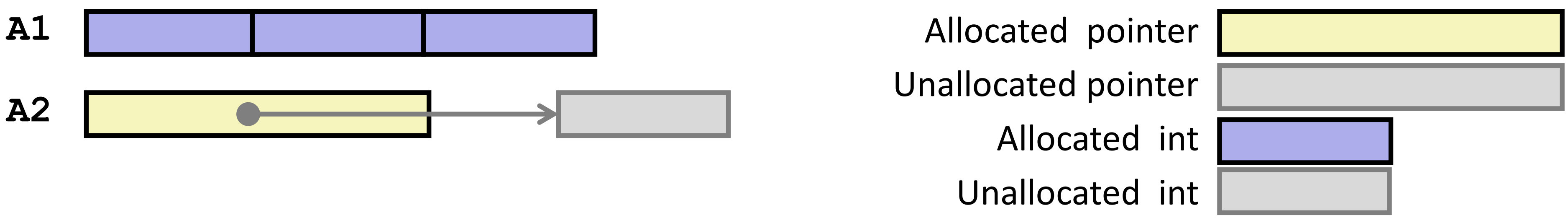

```
... foo(zipcode z, int digit){
   ...
}
```

```
# %rdi = z
# %rsi = digit
leaq (%rdi,%rsi,4), %rax
```

- **Q: What does `foo` return?**
- **A: `&(z[digit]);`**

Understanding Pointers & Arrays #1

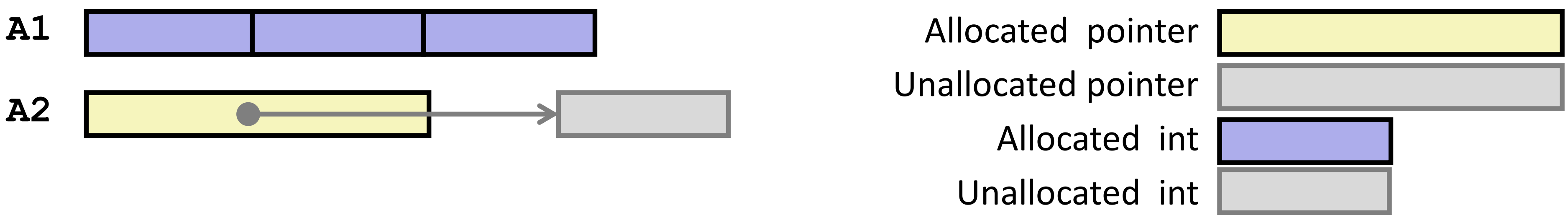
Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						



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

Understanding Pointers & Arrays #1

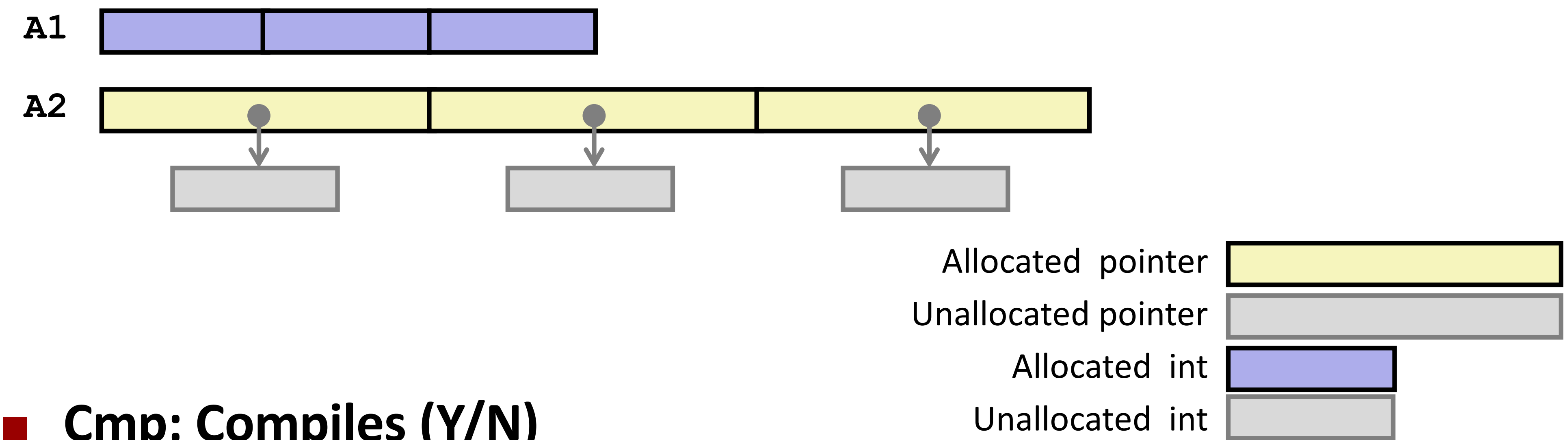
Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4



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

Understanding Pointers & Arrays #2

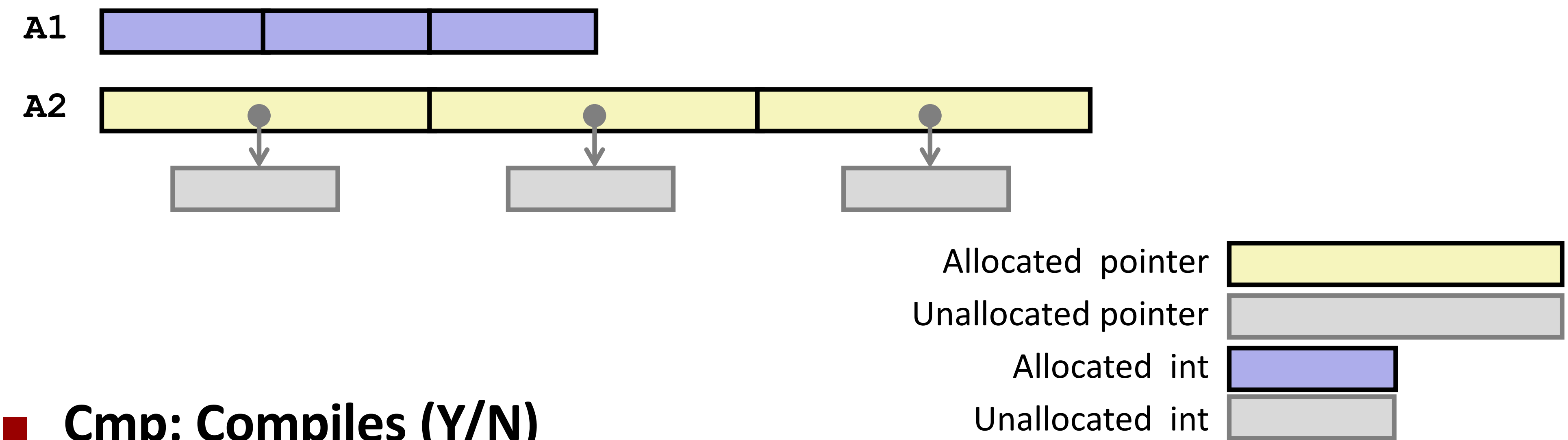
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4			
`int *A2[3]`									



- **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



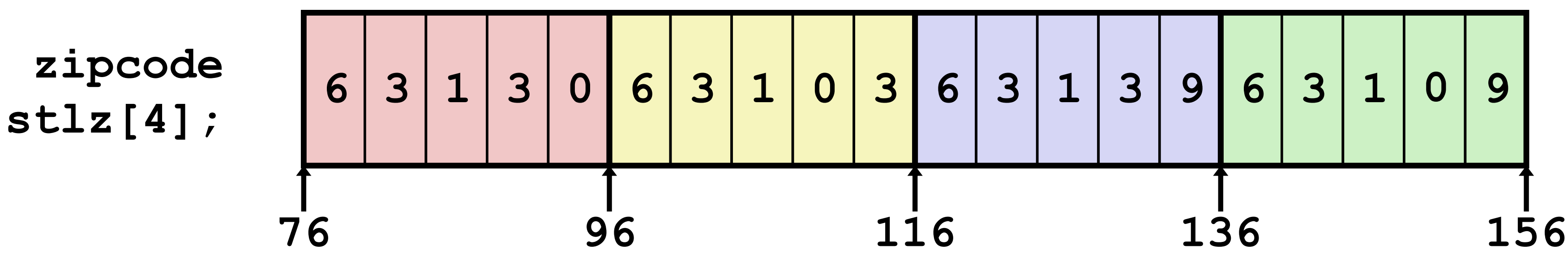
- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Today: Compound Types in C

- **Arrays**
  - One-dimensional arrays
  - Multi-dimensional / nested arrays
  - Multi-level arrays
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Unions**

Nested Array Example

```
#define COUNT 4
typedef int zipcode[ZIP_LEN];
zipcode stlz[COUNT] =
  {{6, 3, 1, 3, 0},
   {6, 3, 1, 0, 3 },
   {6, 3, 1, 3, 9 },
   {6, 3, 1, 0, 9 }};
```



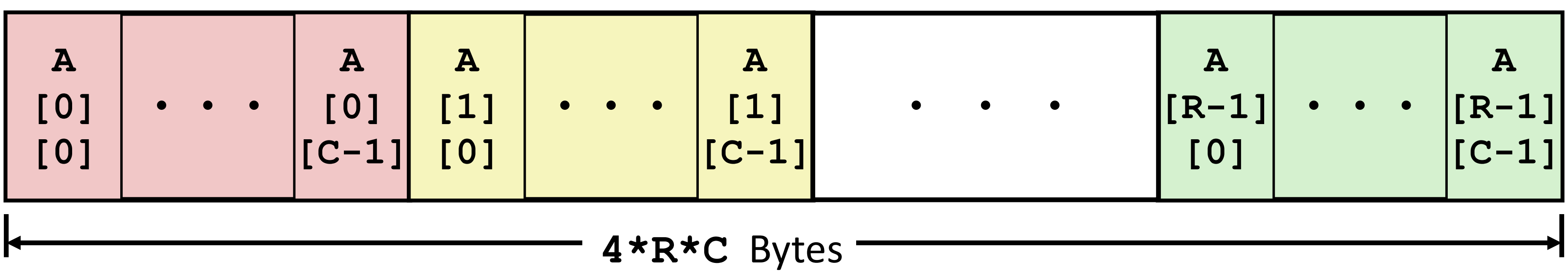
- "`zipcode stlz[4]`" equivalent to "`int stlz[4][5]`"
  - Variable **`stlz`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements guaranteed

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

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

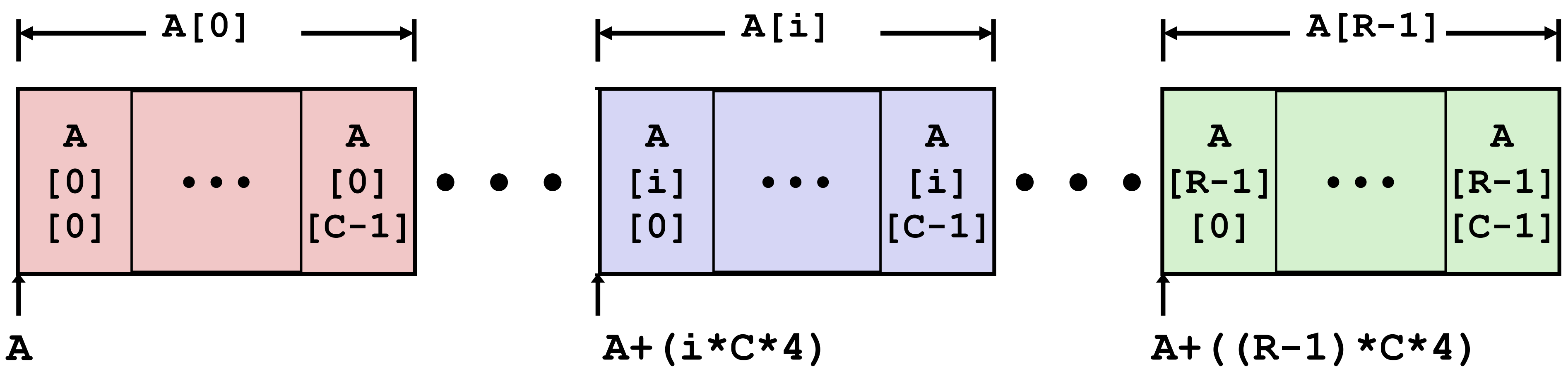
```
int A[R][C];
```



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*)

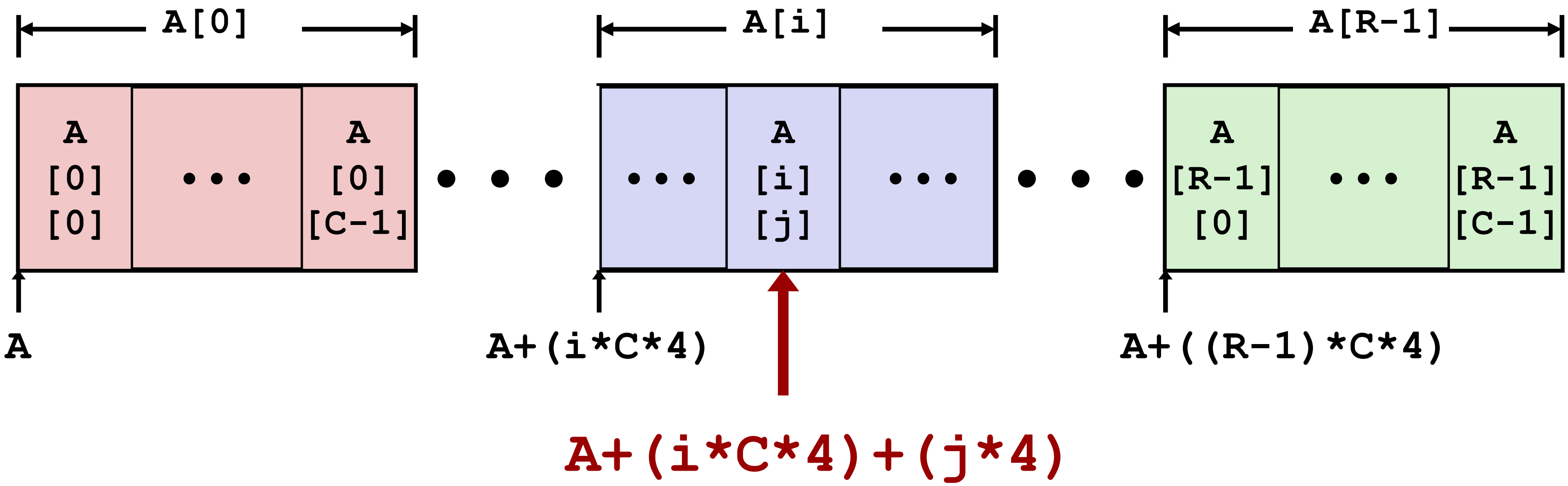
`int A[R][C];`



Nested Array 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];
```

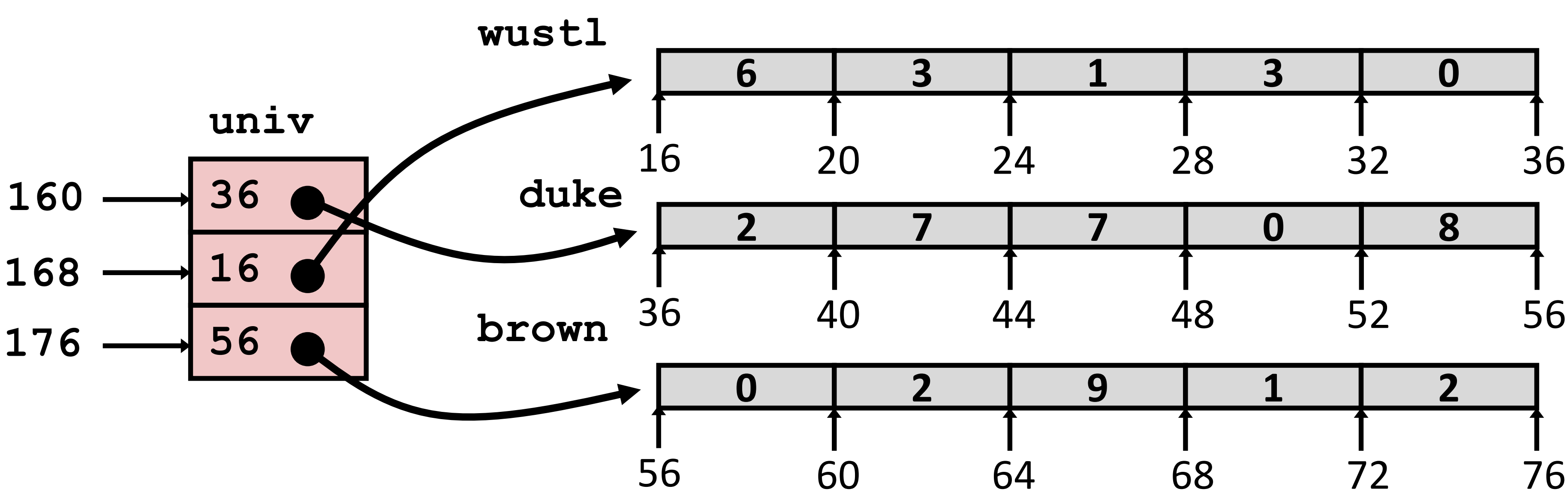


Multi-Level Array Example

```
#define COUNT 4
typedef int zipcode[ZIP_LEN];
zipcode wustl = { 6, 3, 1, 3, 0 };
zipcode duke = { 2, 7, 7, 0, 8 };
zipcode brown = { 0, 2, 9, 1, 2 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {duke,wustl,brown};
```

- 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 *univ, size_t index, size_t digit)
{
  return univ[index][digit];
}
```

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

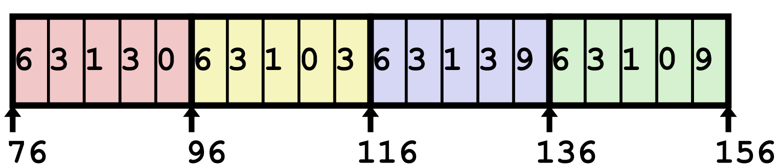
Array Element Accesses

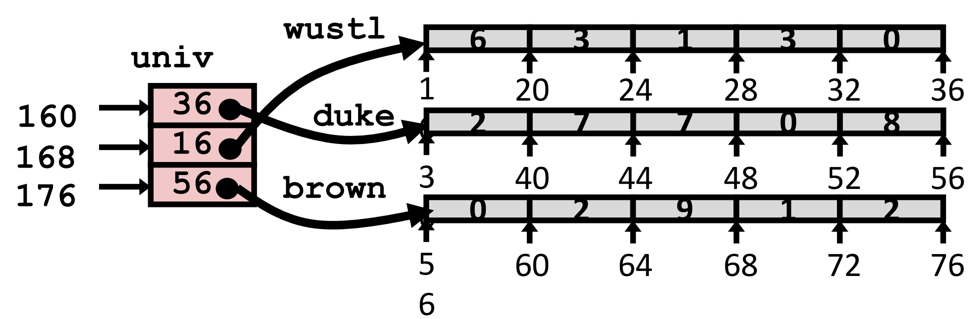
Nested array

```
int get_stl_zip
   (size_t index, size_t digit)
{
  return stlz[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[stlz+20*index+4*digit]` `Mem[Mem[univ+8*index]+4*digit]`

Ex: N X N Matrix Code (2D Nested Array)

- **Variable dimensions, explicit indexing**
  - Traditional way to implement arrays

```c
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j){
  return a[IDX(n,i,j)];
}
```

- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```c
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

- **Fixed dimensions**
  - Know value of N at compile time

```c
#define N 16
/* Get element a[i][j] */
int fix_ele(int a[N][N],
            size_t i, size_t j){
  return a[i][j];
}
```

n X n Matrix Access

- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = n, K = 4
  - Must perform integer multiplication

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t 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
```

16 X 16 Matrix Access

- **Array Elements**
  - Address **A +** $i * (C * K) + j * K$
  - C = 16, K = 4
  - Size known at compile time; may be able to compute index offset more efficiently

```
#define N 16
/* Get element a[i][j] */
int fix_ele(int a[N][N], size_t i, size_t 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
```

Recap: What We Learned Thus Far

- **Nested Array vs Multi-level Array:**
  - Nested array: allocated in contiguous memory, row-major order.
  - Multi-level array: each level uses continuous memory but not across levels.

- **Compiler supports limited form of nested array**
  - The array dimensions must be known to the compiler, either as a compile-time constants or variables.
  - Knowing the array size at compile time may lead to more efficient array index calculation.

Today: Compound Types in C

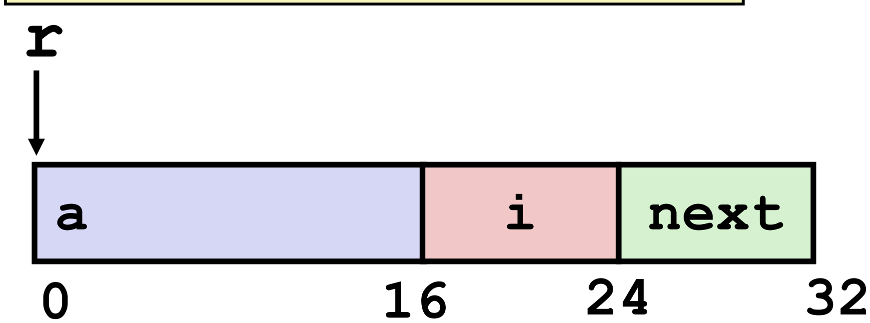
- **Arrays**
  - One-dimensional arrays
  - Multi-dimensional / nested arrays
  - Multi-level arrays
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Unions**

Struct in C

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

or

```
typedef struct rec {
    int a[4];
    size_t i;
    struct rec *next;
} rec_t;
rec_t g;
rec_t *r = &g;
```



- **Concept**
  - Groups data of possibly different types into a single object
  - Refer to members within structure by names
    - r->a[2]
    - g.a[2]