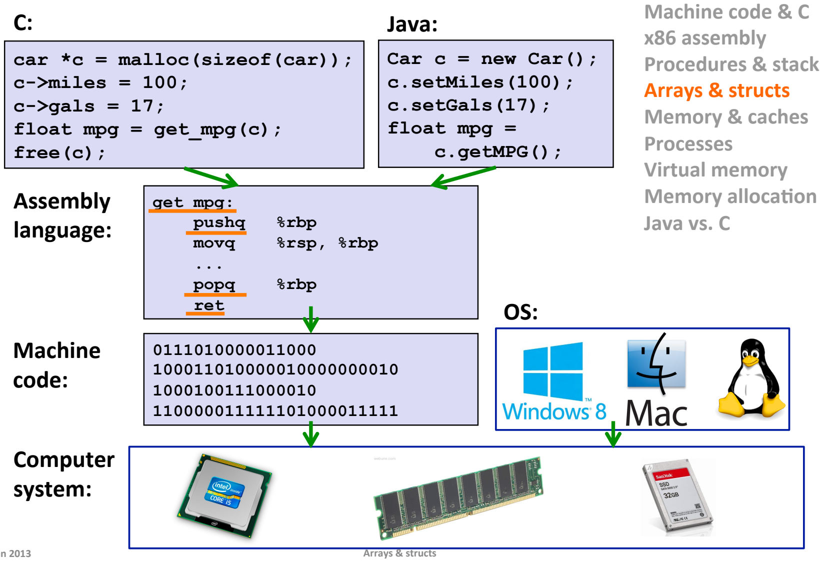
Roadmap

- Memory & data
- Integers & floats
- Machine code & C
- x86 assembly
- Procedures & stacks
- **Arrays & structs**
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C



C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

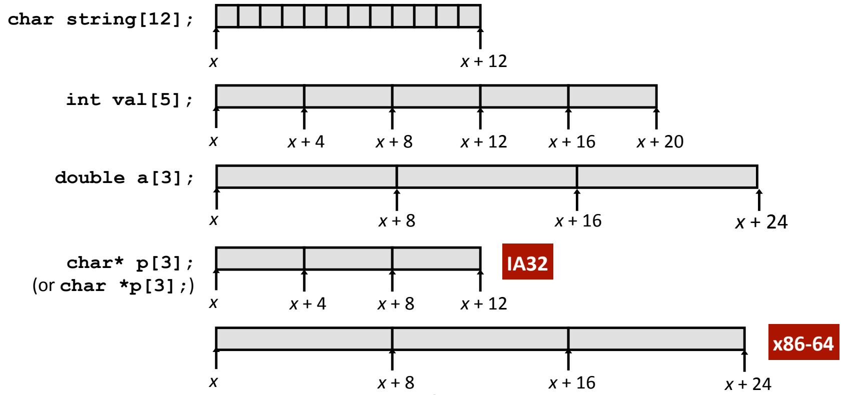
Computer system:

Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structs**
  - Alignment
- **Unions**

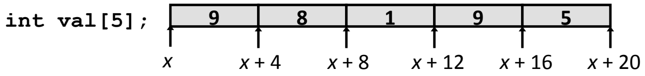
Array Allocation

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - *Contiguously* allocated region of N * sizeof(T) bytes



Array Access

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

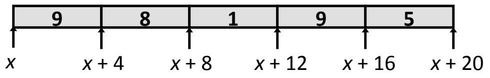


Reference	Type	Value
val[4]	int	
val	int *	
val+1	int *	
&val[2]	int *	
val[5]	int	
*(val+1)	int	
val + i	int *	

Array Access

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

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



Reference	**Type**	**Value**
val[4]	int	5
val	int *	*x*
val+1	int *	*x* + 4
&val[2]	int *	*x* + 8
val[5]	int	?? (whatever is in memory at address *x* + 20)
*(val+1)	int	8
val + i	int *	*x* + 4*i

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

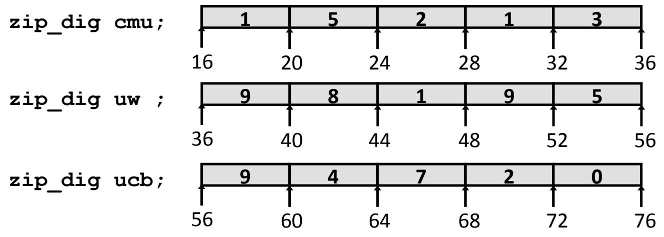
initialization

int uw[5] …

Array Example

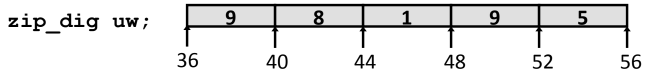
```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- **Declaration “`zip_dig uw`” equivalent to “`int uw[5]`”**
- **Example arrays happened to be allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example



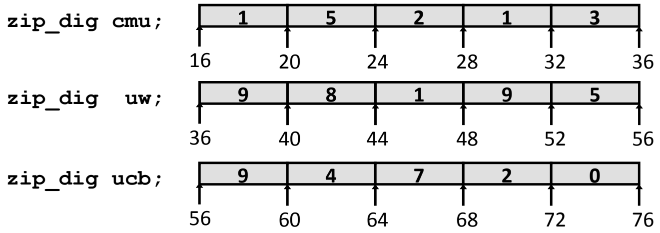
```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax   # z[dig]
```

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

Referencing Examples



Reference	Address	Value	Guaranteed?
`uw[3]`	`36 + 4* 3 = 48`	`9`	Yes
`uw[6]`	`36 + 4* 6 = 60`	`4`	No
`uw[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`16 + 4*15 = 76`	`??`	No

- No bounds checking
- Location of each separate array in memory is not guaranteed

---

Array Loop Example

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

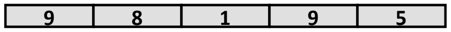
zi = 10*0 + 9 = 9

zi = 10*9 + 8 = 98

zi = 10*98 + 1 = 981

zi = 10*981 + 9 = 9819

zi = 10*9819 + 5 = 98195



Array Loop Example

- **Original**

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- **Transformed**
  - Eliminate loop variable **`i`**, use pointer **`zend`** instead
  - Convert array code to pointer code
    - Pointer arithmetic on **`z`**
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

Array Loop Implementation (IA32)

- **Registers**

```
%ecx  z
%eax  zi
%ebx  zend
```

- **Computations**
  - `10*zi + *z` implemented as `*z + 2*(5*zi)`
  - `z++` increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax          # zi = 0
  leal 16(%ecx),%ebx      # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # zi + 4*zi = 5*zi
  movl (%ecx),%eax        # *z
  addl $4,%ecx            # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx          # z : zend
  jle .L59                # if <= goto loop
```

Nested Array Example

```
zip_dig sea[4] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

Remember, **`T A[N]`** is an array with elements of type **`T`**, with length **`N`**

same as:

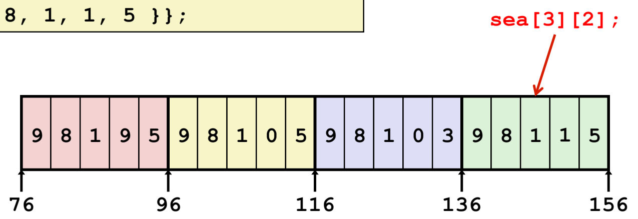
`int sea[4][5];`

What is the layout in memory?

Nested Array Example

```
zip_dig sea[4] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

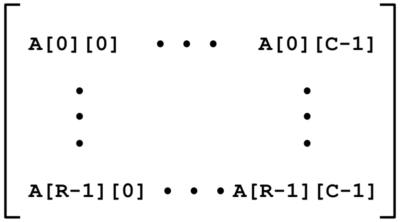
Remember, **`T A[N]`** is an array with elements of type **`T`**, with length **`N`**



- **"Row-major" ordering of all elements**
- **Guaranteed (in C)**

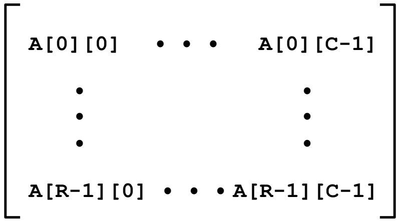
Two-Dimensional (Nested) Arrays

- **Declaration**
  - T A[R][C];
  - 2D array of data type T
  - R rows, C columns
  - Type T element requires K bytes
- **Array size?**

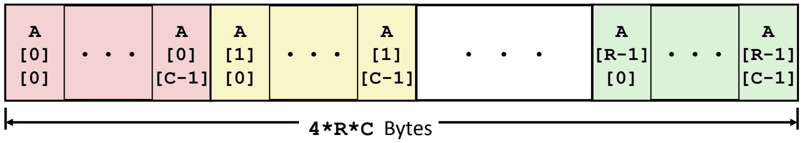


Two-Dimensional (Nested) Arrays

- **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



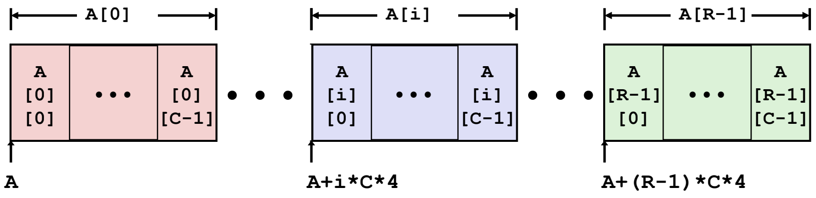
`int A[R][C];`



Nested Array Row Access

- **Row vectors**
  - T   A[R][C]: 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_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

- **What data type is `sea[index]`?**
- **What is its starting address?**

Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

- **What data type is `sea[index]`?**
- **What is its starting address?**

```
 # %eax = index
  leal (%eax,%eax,4),%eax
  leal sea(,%eax,4),%eax
```

Translation?

Nested Array Row Access Code

```
int *get_sea_zip(int index)
{
  return sea[index];
}
```

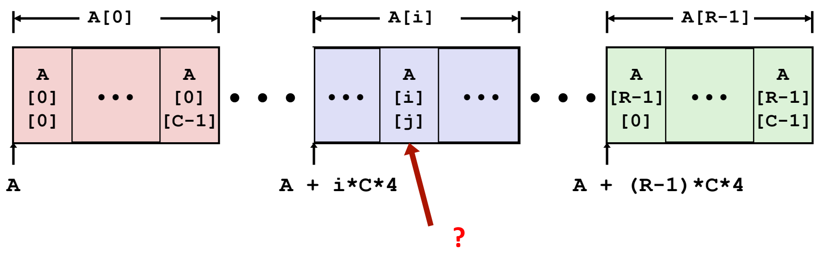
```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
 # %eax = index
  leal (%eax,%eax,4),%eax  # 5 * index
  leal sea(,%eax,4),%eax   # sea + (20 * index)
```

- **Row Vector**
  - `sea[index]` is array of 5 `int`s
  - Starting address `sea+20*index`
- **IA32 Code**
  - Computes and returns address
  - Compute as `sea+4*(index+4*index)=sea+20*index`

Nested Array Row Access

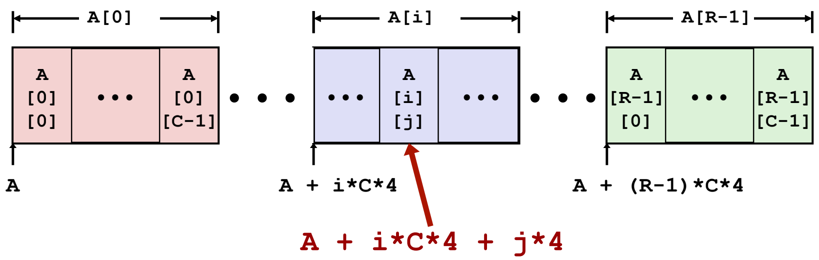
`int A[R][C];`



Nested Array Row 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

```
int get_sea_digit
  (int index, int dig)
{
  return sea[index][dig];
}
```

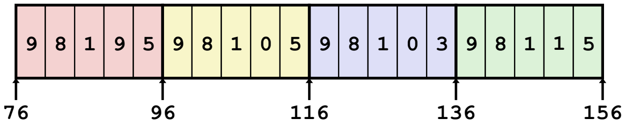
```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (%eax,%eax,4),%eax       # 5*index
movl sea(%edx,%eax,4),%eax    # *(sea + 4*dig + 20*index)
```

- **Array Elements**
  - sea[index][dig] is int
  - Address: sea + 20*index + 4*dig
- **IA32 Code**
  - Computes address sea + 4*dig + 4*(index+4*index)
  - movl performs memory reference

Strange Referencing Examples

```
zip_dig
sea[4];
```



Reference	Address	Value	Guaranteed?
`sea[3][3]`	`76+20*3+4*3 = 148`	1	Yes
`sea[2][5]`	`76+20*2+4*5 = 136`	9	Yes
`sea[2][-1]`	`76+20*2+4*-1 = 112`	5	Yes
`sea[4][-1]`	`76+20*4+4*-1 = 152`	5	Yes
`sea[0][19]`	`76+20*0+4*19 = 152`	5	Yes
`sea[0][-1]`	`76+20*0+4*-1 = 72`	??	No

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

N-dimensional arrays…

double heatMap3D[1024][1024][1024];

total size in bytes?

1024*1024*1024*8 = 8,589,934,592 = roughly 8GB

&heapMap3D[300][800][2] = ?

in bytes: base + 300*1024*1024*8 + 800*1024*8 + 2*8
= base + 8*(2 + 1024*(800+ 1024*(300)))
= base + 2,523,136,016

Multi-Level Array Example

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

```
int univ2D[3] = {
{ 9, 8, 1, 9, 5 },
{ 1, 5, 2, 1, 3 },
{ 9, 4, 7, 2, 0 }
};
```

Same thing as a 2D array?

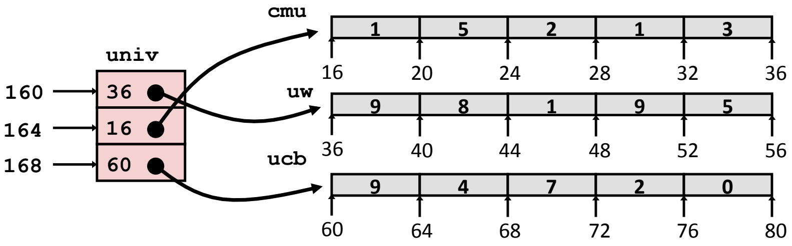
No. One array declaration = one contiguous block of memory.

Multi-Level Array Example

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

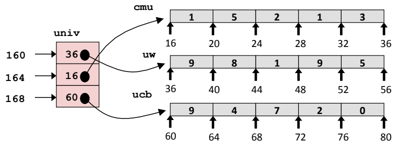
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`s



Note: this is how Java represents multi-dimensional arrays.

Element Access in Multi-Level Array

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



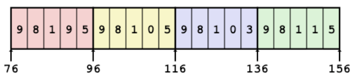
```
  # %ecx = index
  # %eax = dig
  leal 0(,%ecx,4),%edx     # 4*index
  movl univ(%edx),%edx     # Mem[univ+4*index]
  movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

- **Computation (IA32)**
  - Element access `Mem[Mem[univ+4*index]+4*dig]`
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Array Element Accesses

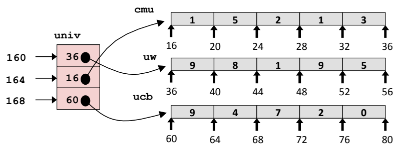
Nested array

```
int get_sea_digit
  (int index, int dig)
{
  return sea[index][dig];
}
```



Multi-level array

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

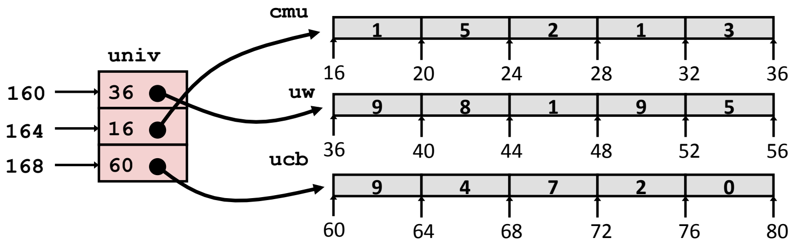


Access looks similar, but it isn't:

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

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

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`60+4*3 = 72`	2	Yes
`univ[1][5]`	`16+4*5 = 36`	9	No
`univ[2][-2]`	`60+4*-2 = 52`	5	No
`univ[3][-1]`	`#@%!^??`	??	No
`univ[1][12]`	`16+4*12 = 64`	4	No

- Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed

---

Using Nested Arrays

```
#define N 16
typedef int fix_matrix[N][N];
```

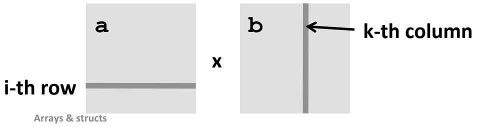
```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```

Using Nested Arrays: arrays of arrays

- **Strengths**
  - Generates very efficient assembly code
  - Avoids multiply in index computation
- **Limitation**
  - Only works for fixed array size

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```



Dynamic Nested Arrays:
arrays of pointers to arrays

- **Strength**
  - Can create matrix of any size
- **Programming**
  - Must do index computation explicitly
- **Performance**
  - Accessing single element costly
  - Must do multiplication

```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
int var_ele
   (int *a, int i, int j, int n)
{
  return a[i*n+j];
}
```

```
movl 12(%ebp),%eax       # i
movl 8(%ebp),%edx        # a
imull 20(%ebp),%eax      # n*i
addl 16(%ebp),%eax       # n*i+j
movl (%edx,%eax,4),%eax  # Mem[a+4*(i*n+j)]
```

Arrays in C

- **Contiguous allocations of memory**
- **No bounds checking**
- **Can usually be treated like a pointer to first element**
- **int a[4][5] => array of arrays**
  - all levels in one contiguous block of memory
- **int* b[4] => array of pointers to arrays**
  - first level in one contiguous block of memory
  - parts anywhere in memory

Structures

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

Memory Layout

- **Characteristics**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structures

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

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator, just like Java:
    - `struct rec r1;  r1.i = val;`
  - What if we have a *pointer* to a struct: `struct rec* r = &r1;`

Structures

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

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator, just like Java:
    - `struct rec r1;  r1.i = val;`
  - What if we have a *pointer* to a struct: `struct rec* r = &r1;`
    - Using `*` and `.` operators: `(*r).i = val;`
    - Or, use `->` operator for short: `r->i = val;`
  - Pointer indicates first byte of structure; access members with offsets

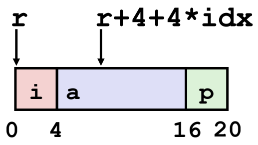
```
void
set_i(struct rec* r,
      int val)
{
     r->i = val;
}
```

IA32 Assembly

```
# %eax = val
# %edx = r
movl  %eax,0(%edx)   # Mem[r+0] = val
```

Generating Pointer to Structure Member

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



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

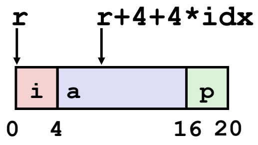
```
int* find_address_of_elem
 (struct rec* r, int idx)
{
   return &r->a[idx];
}
```

`&(r->a[idx])`

```
# %ecx = idx
# %edx = r
leal  0(,%ecx,4),%eax    # 4*idx
leal  4(%eax,%edx),%eax  # r+4*idx+4
```

Generating Pointer to Structure Member

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



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

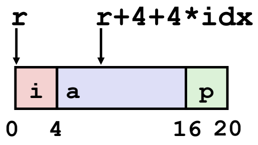
```
int* find_address_of_elem
 (struct rec* r, int idx)
{
   return &r->a[idx];
}
```

`&(r->a[idx])`

```
# %ecx = idx                 OR
# %edx = r
leal  4(%edx,%ecx,4),%eax   # r+4*idx+4
```

Accessing to Structure Member

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



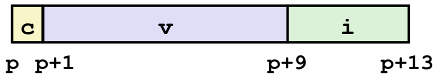
- **Reading Array Element**
  - Offset of each structure member *still* determined at compile time

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

```
# %ecx = idx
# %edx = r
movl  4(%edx,%ecx,4),%eax   # Mem[r+4*idx+4]
```

Structures & Alignment

- **Unaligned Data**

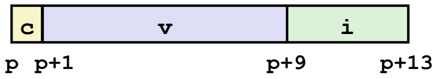


```
struct S1 {
  char c;
  double v;
  int i;
} * p;
```

- **How would it look like if data items were *aligned* *(address multiple of type size)* ?**

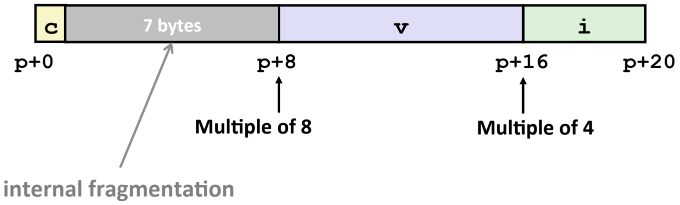
Structures & Alignment

- **Unaligned Data**



```
struct S1 {
  char c;
  double v;
  int i;
} * 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
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, Windows, Mac OS X, …
- **What is the motivation for alignment?**

Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, Windows, Mac OS X, …
- **Motivation for Aligning Data**
  - Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)
    - Inefficient to load or store datum that spans these boundaries
  - Also, virtual memory is very tricky when datum spans two pages (later…)
- **Compiler**
  - Inserts padding in structure to ensure correct alignment of fields
  - `sizeof()` should be used to get true size of structs

Specific Cases of Alignment (IA32)

- **1 byte: char, …**
  - no restrictions on address
- **2 bytes: short, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: int, float, char *, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: double, …**
  - Windows (and most other OSs & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
    - i.e., treated liked 2 contiguous 4-byte primitive data items

---

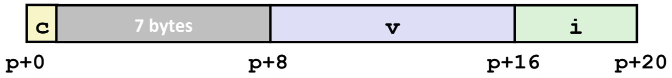
Saving Space

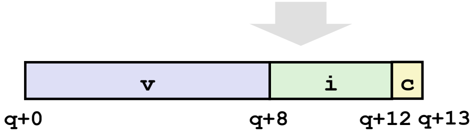
- **Put large data types first:**

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

```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```

- **Effect (example x86-64, both have K=8)**



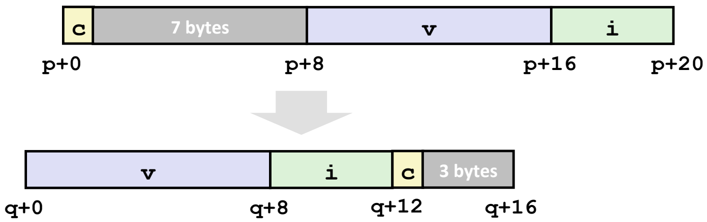


But actually…

Struct Alignment Principles

- **Size must be a multiple of the largest primitive type inside.**

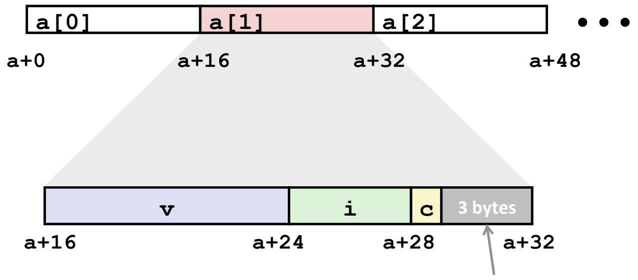
$K = 8$ so $size$ mod $8 = 0$



Arrays of Structures

- **Satisfy alignment requirement for every element**
- **How would accessing an element work?**

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



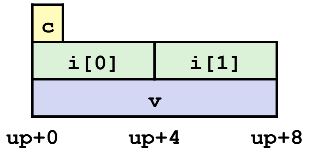
external fragmentation

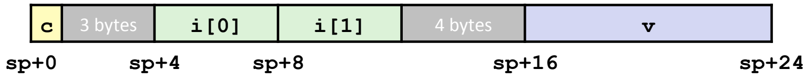
Unions

- Allocated according to largest element
- Can only use one member at a time

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

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





What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different “views” of the same memory location
  - Can be used to circumvent C’s type system (bad idea)
- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts
- But watch out for endianness at a small scale…
- Layout details are implementation/machine-specific…

```
union int_or_bytes {
  int i;
  struct bytes {
       char b0, b1, b2, b3;
  }
}
```

Unions For Embedded Programming

```
typedef union
{
   unsigned char byte;
   struct {
      unsigned char b0:1;
      unsigned char b1:1;
      unsigned char b2:1;
      unsigned char b3:1;
      unsigned char reserved:4;
   } bits;
} hw_register;

hw_register reg;
reg.byte = 0x3F;         // 00111111_2
reg.bits.b2 = 0;         // 00111011_2
reg.bits.b3 = 0;         // 00110011_2
unsigned short a = reg.byte;
printf("0x%X\n", a);    // output: 0x33
```

(Note: the placement of these fields and other parts of this example are implementation-dependent)

Summary

- Arrays in C
  - Contiguous allocations of memory
  - No bounds checking
  - Can usually be treated like a pointer to first element
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Provide different views of the same memory location