Programming Paradigms

Composite Types (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Quiz: Warm-Up

Which (if any) of the following lines lead to a compile-time error in Java?

```
int[] a, b;
int c, d[];

a = new int[2];
d = a;
b = new char[3];
c = new int[4];
```

Quiz: Warm-Up

Which (if any) of the following lines lead to a compile-time error in Java?

```
int[] a, b;
int c, d[];

a = new int[2];
d = a;
b = new char[3];
c = new int[4];
```

Both `a` and `b` are int arrays.

Quiz: Warm-Up

Which (if any) of the following lines lead to a compile-time error in Java?

```
int[] a, b;
int c, d[];

a = new int[2];
d = a;
b = new char[3];
c = new int[4];
```

c is an int,

d is an int array.

Quiz: Warm-Up

Which (if any) of the following lines lead to a compile-time error in Java?

```
int[] a, b;
int c, d[];

a = new int[2];
d = a;
b = new char[3];   ←
c = new int[4];
```

Error 1: char array is incompatible with int array.

Quiz: Warm-Up

Which (if any) of the following lines lead to a compile-time error in Java?

```
int[] a, b;
int c, d[];

a = new int[2];
d = a;
b = new char[3];
c = new int[4];
```

Error 2: Can’t assign int array to int variable.

Composite Types

- **New types formed by joining together simpler types using a type constructor**
- **Common type constructors**
  - Records
  - Arrays
  - Strings
  - Sets
  - Pointers
  - Lists

Overview

- Records ←
- Arrays
- Pointers and Recursive Types

Records

- **A.k.a. structures or structs**
- **Store and manipulate related data of heterogeneous types together**
  - Each data component is a field
- **Originate from**
  - Cobol: Introduced concept
  - Algol 68: Introducted `struct` keyword

Example

A struct in C:

```
struct element {        // defines a record
  char name[2];         // with four fields
  int atomic_number;
  double atomic_weight;
  _Bool metallic;
};
```

Example

A struct in C:

```c
struct element {       // defines a record
  char name[2];        // with four fields
  int atomic_number;
  double atomic_weight;
  _Bool metallic;
};

struct element copper; // variable of record type
copper.name[0] = 'C';
// ...
if (copper.metallic) { // access fields with
    // ...             // dot notation
}
```

Variants Available in Most PLs

Most PLs offer some record-like type constructor

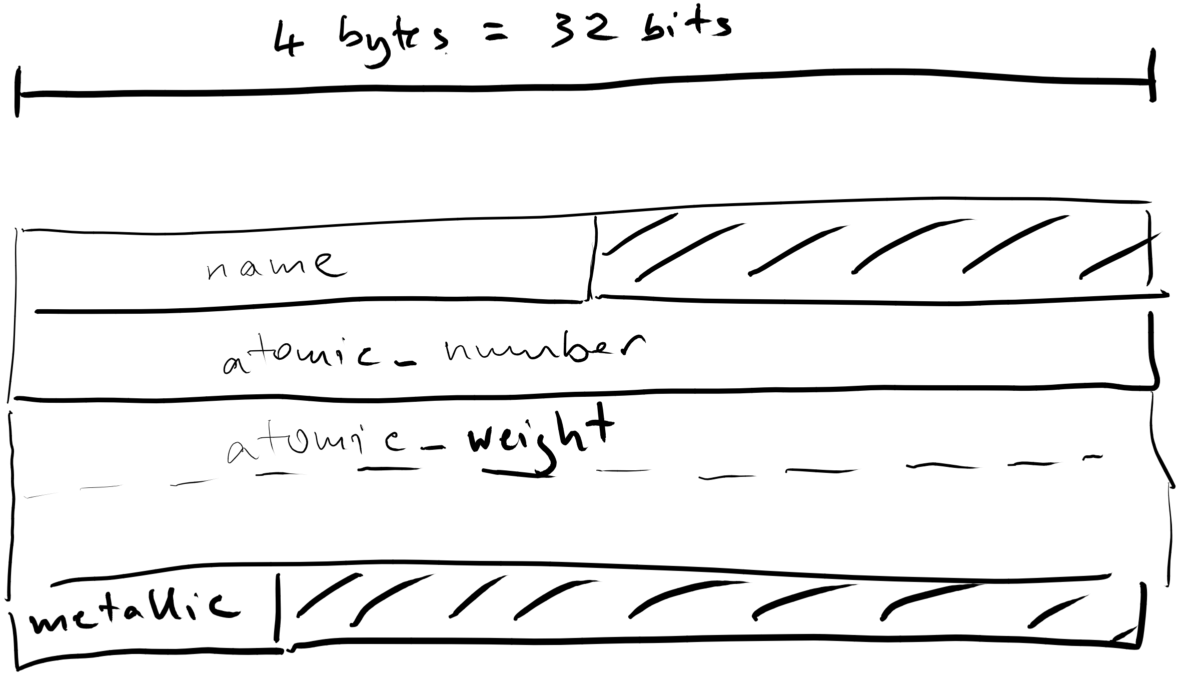
- C: structs
- C++: special form of class
- Fortran 90: simple called “types”
- C#, Swift: struct types (as opposed to class types)
- OCaml: tuples (where order of fields is irrelevant)
- Java: since Java 14, “records” but with immutable fields

Memory Layout

How are records stored in memory?

- Usually, fields stored in adjacent locations
- Field access: Address + offset
- Alignment constraints may create "holes"
  - Alignment constraints depend on architecture
  - E.g., 4-byte ints on x86 must start at address divisible by 4

Example: Memory Layout



Packing and Recording

How to optimize for space?

- Option 1: Packing
  - □ Avoid holes and break alignment
  - □ Will need additional instructions to operate on fields (e.g., to reassemble value into register)
- Option 2: Reordering fields
  - □ Minimize holes but respect alignment

Packing and Recording

How to **optimize for space**?

- Option 1: Packing
  - Avoid holes and break alignment
  - Will need additional instructions to operate on fields (e.g., to reassemble value into register)
- Option 2: Reordering fields
  - Minimize holes but respect alignment

Can instruct compiler to pack a record (e.g., via pragmas in gcc)

Packing and Recording

How to **optimize for space**?

- Option 1: Packing
  - □ Avoid holes and break alignment
  - □ Will need additional instructions to operate on fields (e.g., to reassemble value into register)
- Option 2: Reordering fields
  - □ Minimize holes but respect alignment

System-level programmer may rely on memory layout: C and C++ don’t reorder fields

Quiz: Memory Layout of Records

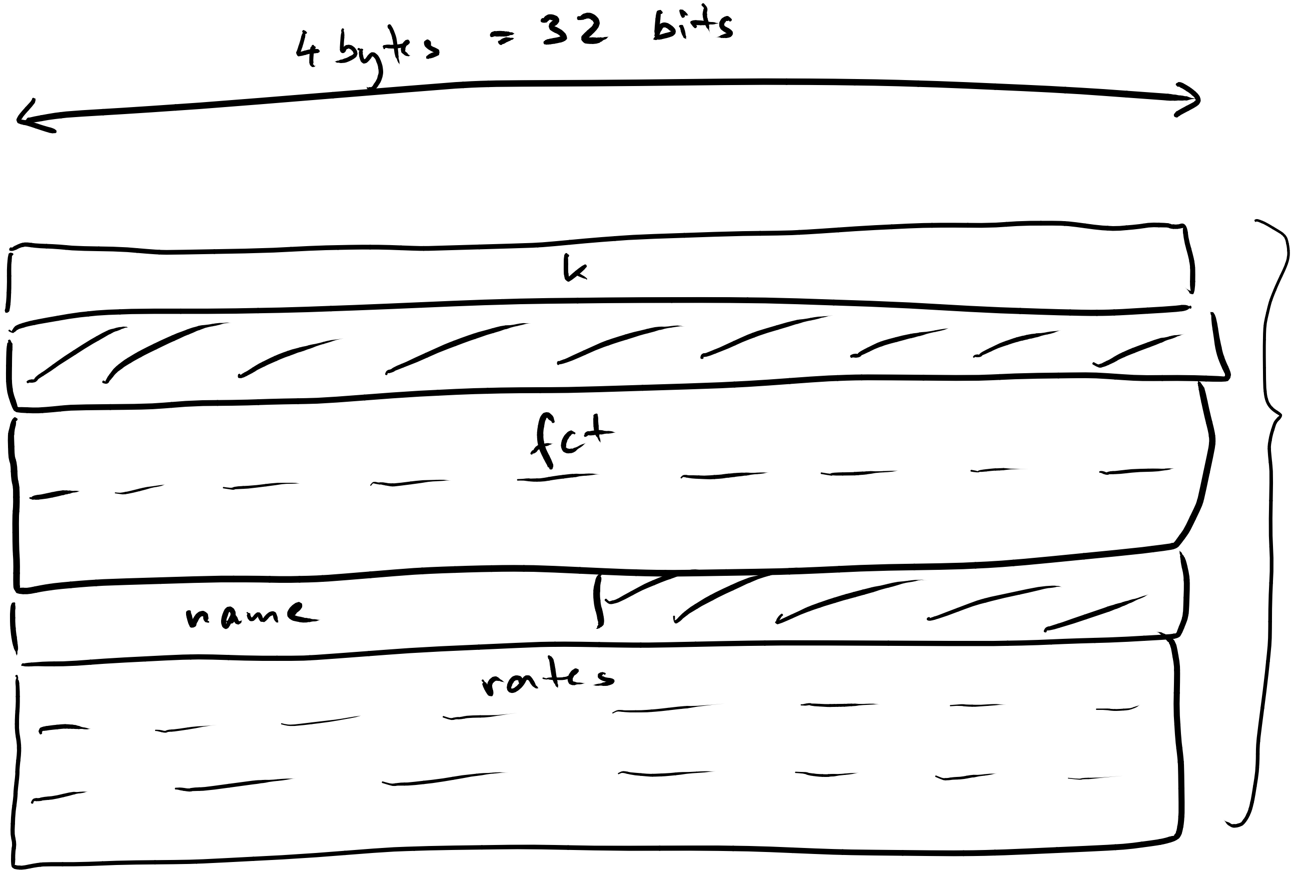
How many bytes does an array of three of the following structs need (without packing)?

```
struct quiz {
    int k;
    void *fct;
    char name[2];
    float rates[3];
};
```

Assumptions:

- Size of char: 1 byte
- Size of int: 4 bytes
- Size of float: 4 bytes
- Size of pointer: 8 bytes
- Floats must be aligned (divisible by 4)
- Pointers must be aligned (divisible by 8)

Quiz: Memory Layout of Records (Corrected)



$4 \cdot 8 = 32$ bytes

$\downarrow \cdot 3$

96 bytes

Quiz: Memory Layout of Records

How many bytes does an array of three of the following structs need (without packing)?

```
struct quiz {
    int k;
    void *fct;
    char name[2];
    float rates[3];
};
```

Tip: Check it yourself with `sizeof(struct quiz)`

Assumptions:

- Size of char: 1 byte
- Size of int: 4 bytes
- Size of float: 4 bytes
- Size of pointer: 8 bytes
- Floats must be aligned (divisible by 4)
- Pointers must be aligned (divisible by 8)

Nested Records

- Option 1: Lexically nested

```
struct outer_record {
  char some_field[10];
  struct { // no name for this inner record
    int some_other_field;
    double yet_another_field;
  } nested_field;
};
```

- Option 2: Fields of record type

```
struct outer_record {
  char some_field[10];
  struct inner_record nested_field;
};
```

Semantics of Nested Records

What's the meaning of referring to a nested record?

```
struct S s1;
struct S s2;
s1.n.j = 0;
s2 = s1;
s2.n.j = 7;
print("%d\n", s1.n.j);
```

Semantics of Nested Records

What's the meaning of referring to a nested record?

```
struct S s1;
struct S s2;
s1.n.j = 0;
s2 = s1;
s2.n.j = 7;
print("%d\n", s1.n.j);
```

Does it print 0 or 7?

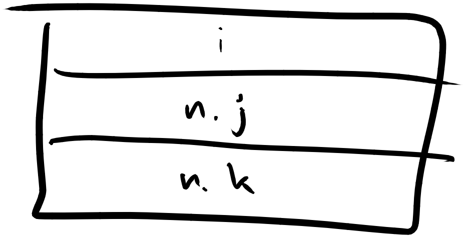
Reference Model vs. Value Model

- **Occurrence of a variable may mean**
  - a reference to its memory location
  - the value stored in the variable
- **E.g., C:**
  - Reference model if variable is left-hand side of assignment
  - Value model otherwise
- **E.g., Java:**
  - Value model only for built-in types

Example

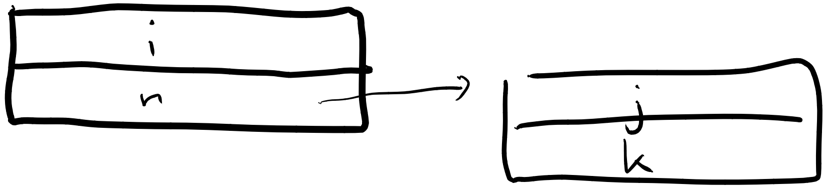
C:

```
struct   T   {
      int j;
      int k;
}
struct   S  {
      int i;
      struct T  n;
}
```



Java:

```
class T {
     public int j;
     public int k;
}
class S {
      public int i;
      public T n;
}
```



Semantics of Nested Records

What’s the meaning of referring to a nested record?

```c
// C code
struct S s1;
struct S s2;
s1.n.j = 0;
s2 = s1;
s2.n.j = 7;
print("%d\n", s1.n.j);
```

```java
// Java code
S s1 = new S();
s1.n = new T();
s1.n.j = 0;
S s2 = s1;
s2.n.j = 7;
System.out.println(s1.n.j);
```

Semantics of Nested Records

What’s the meaning of referring to a nested record?

```
// C code
struct S s1;
struct S s2;
s1.n.j = 0;
s2 = s1;
s2.n.j = 7;
print("%d\n", s1.n.j);
```

Prints 0

```
// Java code
S s1 = new S();
s1.n = new T();
s1.n.j = 0;
S s2 = s1;
s2.n.j = 7;
System.out.println(s1.n.j);
```

Prints 7

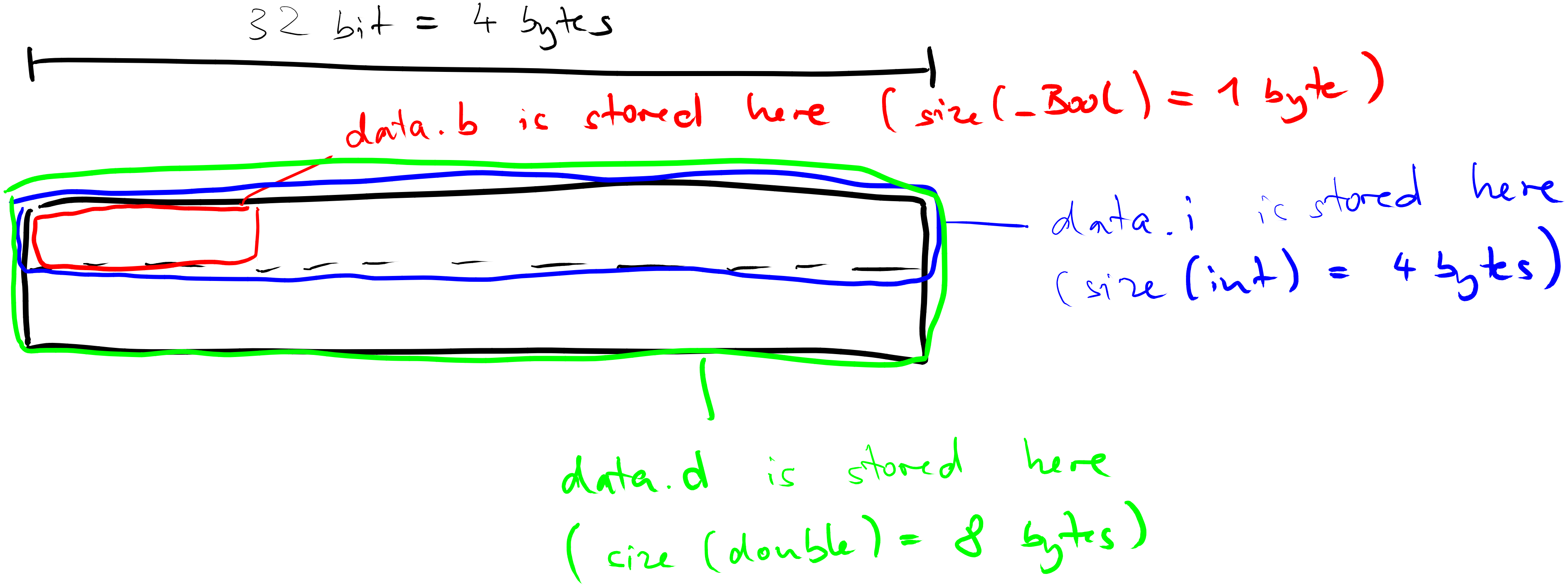
Variant Records (Unions)

- **Special kind of record**
- **Reuses same memory location for multiple variables**
  - Assumption: Variables never used at the same time
  - Size of record = size of largest member

Demo

Demo: union.c

Example: Union in C



Use Cases for Unions

- **Bytes interpreted differently at different times**
  - E.g., implementation of memory manager: Memory blocks contain bookkeeping information and user data
- **Represent single data type with alternative sets of fields**
  - E.g., record for employees: Properties depend on department of employee

Overview

- Records
- Arrays ←
- Pointers and Recursive Types

Arrays

- **Most common composite data type**
- **Conceptually: Mapping from index type to element type**
  - Index types: Usually a discrete type, e.g., integer
  - Element type: Usually any type

Syntax

Varies across PLs

- Declaration
  - C: `char upper[26];`
  - Fortran: `character (26) upper`
- Accessing elements
  - C: `upper[3]` (indices start at 0)
  - Fortran: `upper(3)` (indices start at 1)

Multi-Dimensional Arrays

- **Indexing along multiple dimensions**
  - Single dimension: Sequence of elements
  - Two dimensions: 2D matrix of elements
  - Three dimensions: 3D matrix of elements
  - etc.
- **E.g., two-dimensional array in C:**

```
int arr[3][4];
```

  - 3 rows, 4 columns

Array Operations

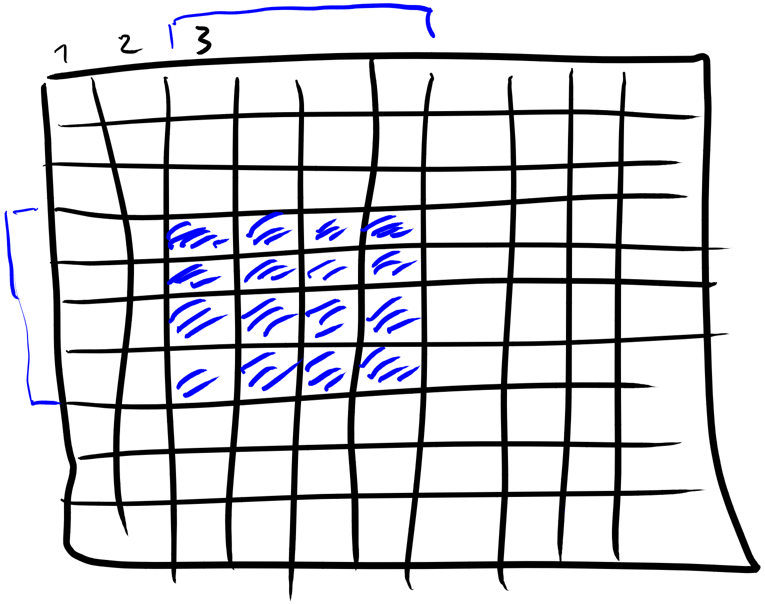
- **Slicing: Extract “rectangular” portion of array**
  - □ Some PLs: Along multiple dimensions
- **Comparison**
  - □ Element-wise comparison of arrays of equal length:

    ```
    arr1 < arr2
    ```
- **Mathematical operations**
  - □ Element-wise addition, subtraction, etc.

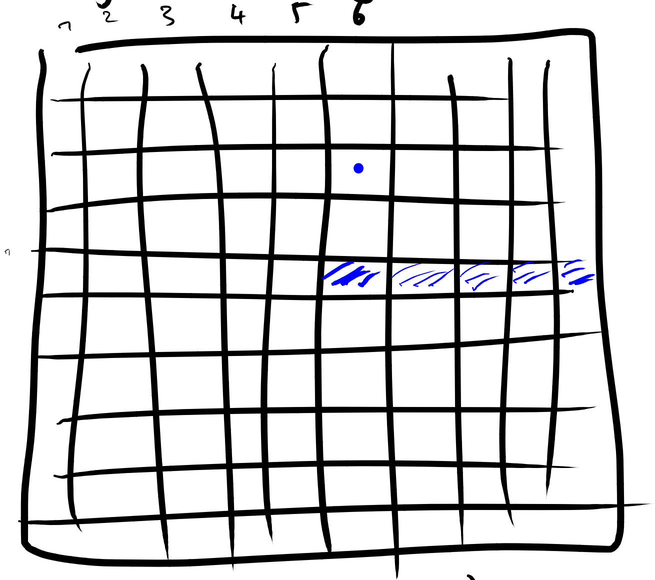
Example: Array Slicing in Fortran

10 × 10 array: matrix

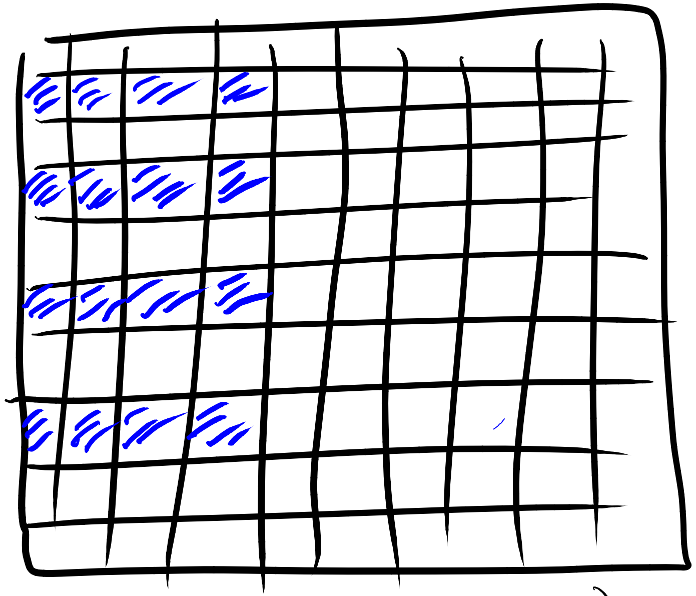
Note: Fortran uses column-major indexing



matrix(3:6, 4:7)



matrix(6:, 5)



matrix(:4, 2:8:2)

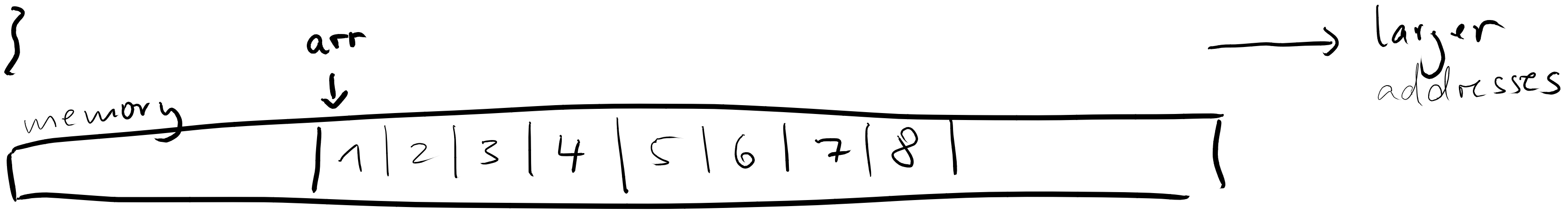
Memory Layout

- **Single dimension: Elements are contiguous in memory**
- **Multiple dimensions**
  - Option 1: Contiguous, row-major layout
    - E.g., in C
  - Option 2: Contiguous, column-major layout
    - E.g., in Fortran
  - Option 3: Row-pointer layout
    - E.g., in Java

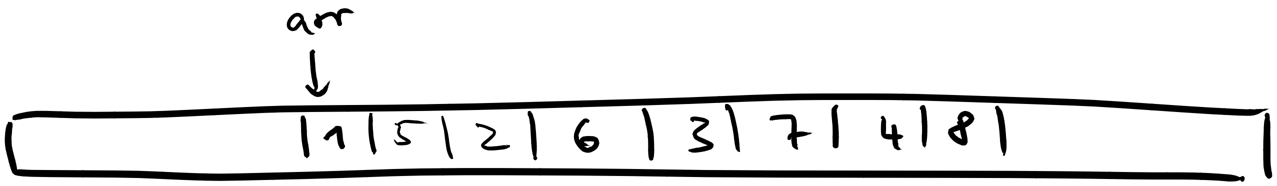
Example

```
int arr [2][4] = {
        {1, 2, 3, 4 },
        { 5, 6, 7, 8 }
}
```

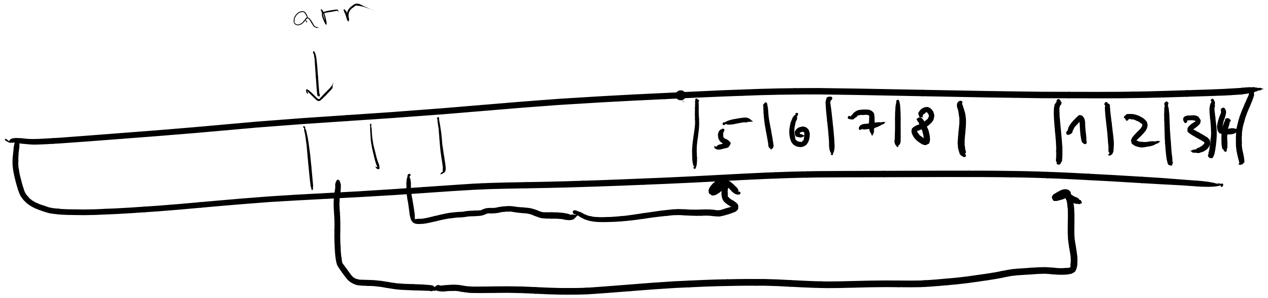
Row-major:



Column-major:



Row pointers:



Significance of Memory Layout

Layout determines efficiency of nested loops that iterate through multi-dimensional arrays

- CPU fetches entire cache lines from memory
- Accessing all data in a cache line is efficient
- Accessing data outside of current cache line: Cache miss
  - Causes expensive reading of another cache line

Quiz: Efficient Array Access

Given a large, two-dimensional array, which loop is faster in C and Fortran?

```c
// C code, option 1
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    // access arr[i][j]
  }
}

// C code, option 2
for (j=0; j<N; j++) {
  for (i=0; i<N; i++) {
    // access arr[i][j]
  }
}
```

```fortran
! Fortran code, option 1
do i=1,N
  do j=1,N
    ! access arr(i,j)
  end do
end do

! Fortran code, option 2
do j=1,N
  do i=1,N
    ! access arr(i,j)
  end do
end do
```

Quiz: Efficient Array Access

Given a large, two-dimensional array, which loop is faster in C and Fortran?

```
// C code, option 1
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    // access arr[i][j]
  }
}
```

```
// C code, option 2
for (j=0; j<N; j++) {
  for (i=0; i<N; i++) {
    // access arr[i][j]
  }
}
```

```
! Fortran code, option 1
do i=1,N
  do j=1,N
    ! access arr(i,j)
  end do
end do
```

```
! Fortran code, option 2
do j=1,N
  do i=1,N
    ! access arr(i,j)
  end do
end do
```

Overview

- Records
- Arrays ✓
- Pointers and Recursive Types