Programming Paradigms

Composite Types (Part 1)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Summer 2020
Which (if any) of the following lines lead to a compile-time error in Java?

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

a = new int[2];
b = new int[3];
c = new char[4];
d = new int[5];
```

Please vote via Ilias.
Quiz

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

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

a = new int[2];
b = new int[3];
c = new char[4];
d = new int[5];
```

a is an int array,  
b is just an int variable.
Quiz

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

```java
int a[], b;
int[] c, d; // Both c and d are int arrays.
a = new int[2];
b = new int[3];
c = new char[4];
d = new int[5];
```

Please vote via Ilias.
Quiz

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

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

a = new int[2];
b = new int[3];
c = new char[4];
d = new int[5];
```

Error 1: Can’t assign int array to int variable.

Please vote via Ilias.
Quiz

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

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

a = new int[2];
b = new int[3];
c = new char[4];
d = new int[5];
```

Error 2: char array is incompatible with int array.

Please vote via Ilias.
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**
- **Store and manipulate related data of heterogeneous types together**
  - Each data component is a **field**
- **Originate from**
  - Cobol: Introduced concept
  - Algol 68: Introduced **struct** keyword
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;
};
```
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: (no records/structs provided)
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

4 bytes / 32 bits

- name
- atomic_number
- atomic_weight
- metallic
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/C++ don’t reorder fields
Quiz: Memory Layout of Records

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

```c
struct quiz {
    int k;
    float rates[3];
    char name[6];
    void *fct;
};
```

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

Please vote via Ilias.
\textbf{Quit:}

4 bytes / 32 bits

\begin{align*}
8 \times 4 &= 32 \text{ bytes} \\
32 \times 5 &= 160 \text{ bytes}
\end{align*}
Quiz: Memory Layout of Records

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

```c
struct quiz {
    int k;
    float rates[3];
    char name[6];
    void *fct;
};
```

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

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

```c
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?

```c
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?
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 ji;
    int hj;
};

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

Java:

```java
class T {
    public int ji;
    public int hj;
};

class S {
    public int i;
    public T n;
};
```
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);

// 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);
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);

// 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 0  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
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