Types (Objectives)

- The student will be able to define type equivalence, type compatibility and type inference.
- Given two types, the student will be able to determine if they are name equivalent or structure equivalent.
- Given a two-dimensional array, the student will be able to compute addresses using row-major, column-major and row-pointer layout.
- Given a structure definition, the student will be able to lay out that structure in memory.

1

Purposes of Types

- Types provide implicit context for many operations, so that the programmer does not have to specify the context explicitly
  - arithmetic operations
  - pointer creation with new
- Types limit the set of operations that may be performed in a semantically valid program
  - no adding of characters to structures
  - cannot invoke an array of integers
  - catch as many errors as possible

2

Type Systems

- High-level languages associates types with values, hardware does not explicitly do so (any type may be stored in any location)
- A type system consists of
  1. a mechanism to define types and associate them with certain language constructs
  2. a set of rules for type equivalence, type compatibility and type inference.
     - Type equivalence – rules to determine when the types of two values are the same
     - Type compatibility – rules to determine when a value of a given type can be used in a particular context
     - Type inference – rules to define the type of an expression based on the types of its constituent parts or surrounding context

3

Type Checking

- Type checking is the process of ensuring that a program obeys the a language's type compatibility rules
  - a violation of the rule is called a type clash
- Definitions
  - strongly typed language – a language that prohibits, in a way that the language implementation can enforce, the application of any operation to any object that is not intended to support the operation
    - Ada
    - Pascal (mostly)
  - statically typed language – strongly typed and type checking is done at compile time
    - Pascal
    - C89
  - dynamically typed – type checking is done at run-time
    - Scheme
  - many languages are a mixture
  - polymorphism – a single body of code operates over objects of multiple types.
    - Scheme and Lisp
    - scripting languages

4

Classification of Types

1. Numeric types
   - integers, floats, etc.
   - precision?
     - some leave it to implementation
     - C and Fortran allow specific declarations
     - Scheme implements
       - integers of arbitrary precision
       - exact rationals
       - floating-point nums that are implementation dependent
2. Enumeration types
   - a set of named elements

     typedef enum {sun, mon, tue, wed, thu, fri, sat} day;

   - implemented as a small set of integers

5

Classification of Types

3. Subrange types
   - a contiguous subset of value from some discrete type
4. Composite types
   - Records (structures)
   - Variant records (unions)
   - Arrays
   - Sets
   - Pointers
   - Lists
   - Files
   - Objects
5. Function types

- Orthogonality is important

6

Type Equivalence

- Given the following type declarations, are they equivalent?

```
struct s1 {
      int a;
      int b;
}

struct s2 {
      int a,b;
}

struct s3 {
      int b;
      int a;
}
```

7

Type Equivalence

- Two types of equivalence
  1. structure equivalence – types of the same structure are equivalent
  2. name equivalence – types of the same name are equivalent

- Another example

```
type a1 = array [1..10] of integer
type a2 = array [0..9] of integer
```

8

Strict Name Equivalence

- Are the types of v1 and v2 equivalent?

```
struct s1 { int a,b; };
typedef s2 s1;

struct s1 v1;
s2  v2;
```

  Does the programmer intend the names to be equivalent?
- Ada

```
subtype stack_element is integer;       /* same */
type stack_element is new integer;     /* different */
```

9

Type Conversion and Casts

- Explicit type conversion (cast) is necessary in one of three cases
  - The types are structurally equivalent and the language uses name equivalence
    - purely conceptual operation
  - The types have different sets of values, but the intersecting values have the same representation
    - must ensure the original type has a value in the converted type
  - The types have different low-level representations, but there is a correspondence between values in both types
    - must convert to new representation (e.g., int to float)
- Non-converting casts do not change the representation of the low-level bits
  - cast the result of malloc in C
  - In Ada, there is an explicit subroutine

```
function cast_float_to_int is new unchecked_conversion(float,integer);
n = cast_float_to_int(f);
```

10

Type Compatibility

- Most languages allow types to be mixed in certain contexts with implicit type conversion (coercion).

```
float f; int n;
f = n + 3.0;
```

- Many languages have a reference type that is compatible with every other reference type.
  - Java → Object
  - C, C++ → void * (see the list routines provided for SomeLife)

11

Arrays

- Arrays are the most common composite data type and have been around since Fortran I
- How should we lay out memory for an array?

```
int a[10];
VAR b: ARRAY [3..12] OF INTEGER;
```

  Should the layout be any different?

12

Computing an Array Address

- In general, for A[i], declared as A[low..high], generate

  base(A) + (i-low)*sizeof(A[low])

Handling One-Dimensional Arrays

- How do we compute the address of b[i]?

VAR i: INTEGER;
b: ARRAY [3..12] OF INTEGER;

Assume i and b are globals.

Relative to $gp, i is stored at offset 0 and b[3] is stored at offset 4 and so on

base address of b is $gp+4

```
# access b[i]
    add $s0, $gp, 0
    lw $s1, 0($s0)
    add $s0, $gp, 4
    sub $s1, $s1, 3
    sll $s1, $s1, 2
    add $s0, $s0, $s1
    lw $s1, 0($s0)
```


Two-dimensional Arrays

- Given A[low_1..$high_1$,low_2..$high_2$], how do we generate code of A[i_1,i_2]?
- Depends on how data is stored
- Consider A[1..2,1..4]

A[1,1]	A[1,2]	A[1,3]	A[1,4]
A[2,1]	A[2,2]	A[2,3]	A[2,4]

Two-dimensional Arrays

- Row-major order – C, C++

A[1,1]	A[1,2]	A[1,3]	A[1,4]	A[2,1]	A[2,2]	A[2,3]	A[2,4]

- Column-major order - Fortran

A[1,1]	A[2,1]	A[1,2]	A[2,2]	A[1,3]	A[2,3]	A[1,4]	A[2,4]

Two-dimensional Arrays

- Row-major order

  base(A) + ((i_1 – low_1) * ($high_2$-low_2+1) + i_2-low_2)*sizeof(A[1][1])

- Column-major order

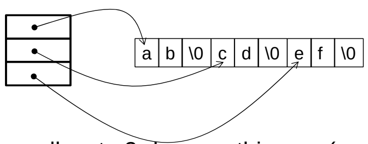
  base(A) + ((i_2 – low_2) * ($high_1$-low_1+1) + i_1-low_1)*sizeof(A[1][1])

17

Row –Pointer Layout

- In C, a 2-d array may be allocated as a single-dimension array of pointers to single-dimension arrays.

  char *[] = { “ab”, “cd”, “ef”};



Java allocate 2-d arrays this way (space need not be consecutive for second dimension).

18

Practice Problem

- Give assembler to compute the address of the given array element using row-major, column-major and row-pointer layout

  VAR a : ARRAY [3..12][0..9] OF INTEGER;

  = a[5][7]

19

Structures

- Structures
  - usually laid out contiguously
  - possible holes for alignment reasons
  - smart compilers may re-arrange fields to minimize holes (C compilers promise not to)

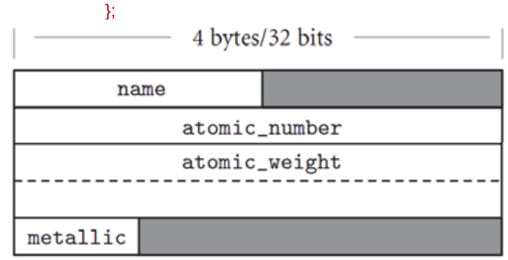
20

Unions

- Unions (variant records)
  - overlay space
  - cause problems for type checking
- Lack of tag means you don't know what is there
- Ability to change tag and then access fields hardly better
  - can make fields "uninitialized" when tag is changed (requires extensive run-time support)
  - can require assignment of entire variant, as in Ada

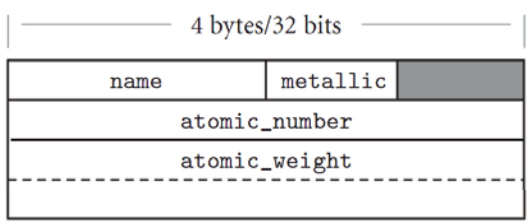
Example Structure & Layout

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    byte metallic;
};
```



Structures

- Rearranged layout (illegal in C)

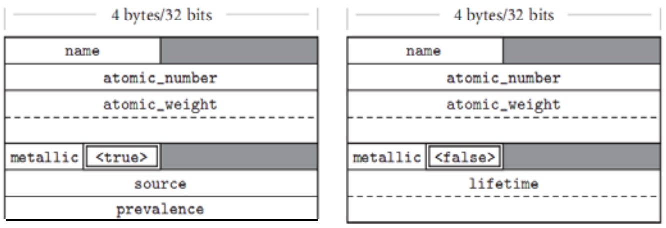


Example Union

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    byte metallic;
    union {
        struct t_data {
          int source;
          int prevalence;
        }                data;
        int lifetime;
    }
};
```

Unions

- Memory layout and its impact (unions)



25