The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts

Spring 2003

Type Checking and Data Type Implementation

Stotts, Hernandez-Campos

1

Type Checking

- Type equivalence
  - *When are the types of two values the same?*
- Type compatibility
  - *When can a value of type A be used in a context that expects type B?*
- Type inference
  - *What is the type of an expression, given the types of the operands?*

2

Type Checking

- **Type equivalence**
  - ***When are the types of two values the same?***
- Type compatibility
  - *When can a value of type A be used in a context that expects type B?*
- Type inference
  - *What is the type of an expression, given the types of the operands?*

3

Type Equivalence

- Type equivalence is defined in two principal ways
  - Structural equivalence
  - Name equivalence
- Two types are **structurally equivalent** if they have identical type structures
  - They must have the same components
  - *E.g.* C
- Two type are **nominally equivalent** if they have the same name
  - It may or may not include alias
  - *E.g.* Java

4

Type Equivalence
Structural Equivalence

- Does the format of the declaration matter?

```
typedef struct { int a, b; } foo1;
```

```
typedef struct {
        int a, b;
} foo2;
```

- All languages consider these two types structurally equivalent

5

Type Equivalence
Structural Equivalence

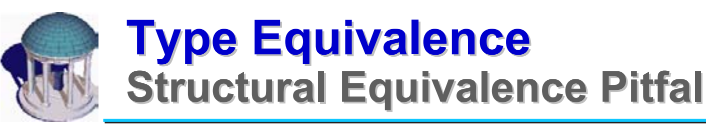
- Is the order in the declaration relevant?

```
typedef struct {
    int a;
    int b;
} foo1;
```

```
typedef struct {
    int b;
    int a;
} foo2;
```

- Most languages consider these two types structurally equivalent

6



Type Equivalence
Structural Equivalence Pitfall

- Are these two types structurally equivalent?

```
typedef struct {
     char *name;
     char *address;
     int age;
} student;
```

```
typedef struct {
     char *name;
     char *address;
     int age;
} school;
```

- They are, and it is unlikely that the programmer intended to make these two types equivalent

7

Type Equivalence
Name Equivalence

- Assumes type definitions with different names are not equivalent
  – Otherwise, why did the programmer create two definitions?
- It solves the previous problem
  – `student` and `school` are not nominally equivalent
- Aliases:
  – Under *strict name equivalence*, aliases are not equivalent
    » `type A = B` is a definition (*i.e.* new object)
  – Under *loose name equivalence*, aliases are equivalent
    » `type A = B` is a declaration (*i.e.* binding)

8

Type Equivalence
Name Equivalence and Aliases

- Loose name equivalence may introduce undesired type equivalences

```
TYPE celsius_temp = REAL;
     fahrenheit_temp = REAL;
VAR  c : celsius_temp;
     f : fahrenheit_temp;
...
f := c;           (* this should probably be an error *)
```

9

Type Conversion

- A values of one type can be used in a context of another type using *type conversion* or *type cast*
- Two flavors:
  - *Converting type cast*: underlying bits are changed
    - » *E.g.* In C,

```
int i; float f = 3.4 ;
i = (int) f ;  /* at runtime  */
```

  - *Nonconverting type cast*: underlying bits are not altered
    - » *E.g.* In C,

```
int i; float f = 3.4;
i = *((int *) &f);   /* at compile time */
```

10

Type Checking

- Type equivalence
  - *When are the types of two values the same?*
- **Type compatibility**
  - ***When can a value of type A be used in a context that expects type B?***
- Type inference
  - *What is the type of an expression, given the types of the operands?*

11

Type Compatibility

- Most languages do not require type equivalence in every context
- Two types T and S are compatible in Ada if any of the following conditions is true:
  - T and S are equivalent
  - T is a subtype of S
  - S is a subtype of T
  - T and S are arrays with the same number of elements and the same type of elements

12

Type Compatibility

- Implicit type conversion due to type compatibility are known as *type coercions*
  - They may involved low-level conversions
- Example

```
d : weekday;         -- as above
k : workday;         -- as above
type calendar_column is new weekday;
c : calendar_column
...
k := d;      -- run-time check required
d := k;      -- no check required; every workday is a weekday
c := d;      -- static semantic error;
             -- weekdays and calendar_columns are not compatible
```

13

Type Compatibility

- Type coercions make type systems weaker
- Example:

```
short int s;
unsigned long int l;
char c;     /* may be signed or unsigned -- implementation-dependent */
float f;    /* usually IEEE single-precision */
double d;   /* usually IEEE double-precision */
...
s = l;  /* l's low-order bits are interpreted as a signed number. */
l = s;  /* s is sign-extended to the longer length, then
           its bits are interpreted as an unsigned number. */
s = c;  /* c is either sign-extended or zero-extended to s's length;
           the result is then interpreted as a signed number. */
f = l;  /* l is converted to floating-point.  Since f has fewer
           significant bits, some precision may be lost. */
d = f;  /* f is converted to the longer format; no precision lost. */
f = d;  /* d is converted to the shorter format; precision may be lost.
           If d's value cannot be represented in single-precision, the
           result is undefined, but NOT a dynamic semantic error. */
```

14

Type Compatibility

- Generic container objects require a *generic reference type*
- Example

```
import java.util.*;      // library containing Stack container class
...
Stack my_stack = new Stack();
String s = "Hi, Mom";
foo f = new foo ();      // f is of user-defined class type foo
...
my_stack.push (s);
my_stack.push (f);       // we can push any kind of object on a stack
...
s = (String) my_stack.pop();
    // type cast is required, and will generate an exception at run
    // time if element at top-of-stack is not a string
```

15

Type Checking

- Type equivalence
  - *When are the types of two values the same?*
- Type compatibility
  - *When can a value of type A be used in a context that expects type B?*
- **Type inference**
  - ***What is the type of an expression, given the types of the operands?***
  - Generally easy, but subranges and composites may complicate this issue

16

Records

In Pascal, a simple record might be defined as follows:

```
type two_chars = packed array [1..2] of char;
    (* a 'packed' array of char is compatible with a quoted string *)
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```

In C, the corresponding declaration would be

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    char metallic;          /* C has no Boolean type */
};
```

17

Records
Memory Layout

- Basic layout in 32-bit machines
  - There may be *holes* in the allocation of memory

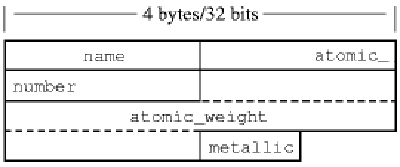
4 bytes/32 bits

name	(hole)
atomic_number	
atomic_weight	
metallic	(hole)

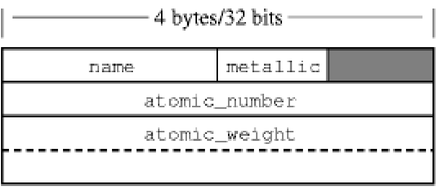
18

Records
Memory Layout

- Packed layout



- Rearranging record field



19

Records
Implications of Memory Layout

- More memory efficient layouts have several drawbacks
  - Assignments require multiple instructions
    - » Masking and shifting operations
  - Access to record elements requires multiple instructions
    - » Masking and shifting operations
- Holes complicate record equality
  - Requires field by field comparison or default values in holes
  - Some languages forbid record comparison
    - » *E.g.* Pascal, C

20

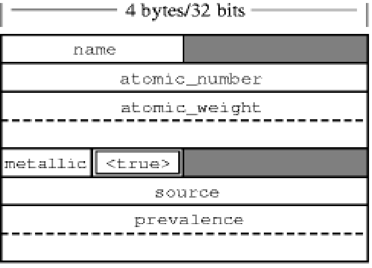
Variant Records

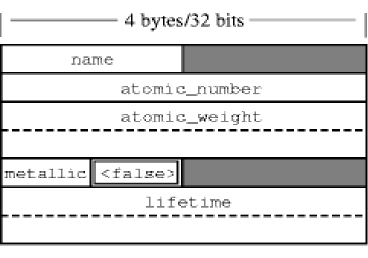
- A variant record provides two or more alternative fields or collections of fields, **but** only one bit is valid at any given time
  - They are called unions in C/C++

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    char metallic;
    char naturally_occurring;
    union {
        struct {
            char *source;
            double prevalence;
        } natural_info;
        double lifetime;
    } extra_fields;
} copper;
```

21

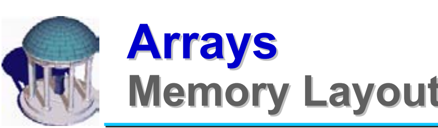
Variant Records
Memory Layout





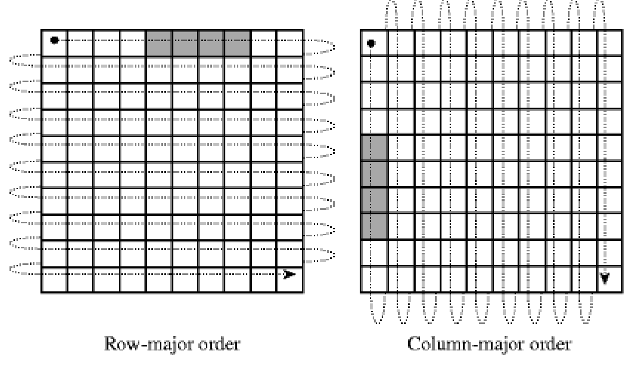
- Some languages, like C, do not check whether variant records are properly access
- Other languages keep track of the value in an additional field, increasing safety

22



Arrays
Memory Layout

- Arrays are usually stored in contiguous locations
- Two common orders



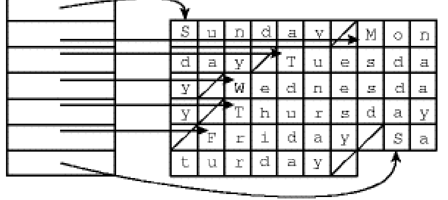
Arrays
Memory Layout

- Contiguous allocation vs. row pointers

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

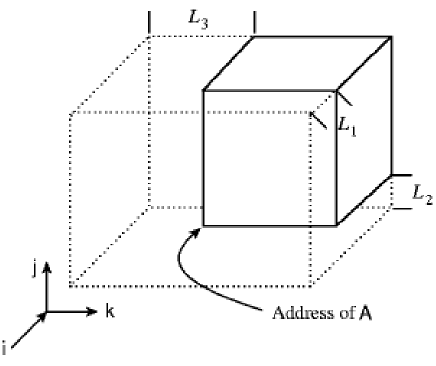
```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

S	u	n	d	a	y				
M	o	n	d	a	y				
T	u	e	s	d	a	y			
W	e	d	n	e	s	d	a	y	
T	h	u	r	s	d	a	y		
F	r	i	d	a	y				
S	a	t	u	r	d	a	y		



Arrays
Address Calculations

- Code generation for a 3D array
  - A: array [L_1..U_1] of array [L_2..U_2] of array [L_3..U_3] of elem_type



25

Arrays
Address Calculations

define

$$S_3 = \text{size of elem_type}$$
$$S_2 = (U_3 - L_3 + 1) \times S_3$$
$$S_1 = (U_2 - L_2 + 1) \times S_2$$

The address of A[i, j, k] is

$$\text{address of A} + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3$$

Row-Major Order

Optimization

$$= (i \times S_1) + (j \times S_2) + (k \times S_3) + \text{address of A} - [(L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)]$$

Compile-Time Constant

26

Reading Assignment

- Scott's chapter 7
  - Section 7.2 except 7.2.4 and 7.2.5
  - Section 7.3 except 7.3.3
  - Section 7.4.3

27