# Chapter 7

## ***Data types***

February 23, Lecture 11

## Types

- Simple types
  - Booleans (or logicals) : 1 byte
  - C lacks booleans
  - Characters: 1 byte traditionally, 2 bytes for Unicode

- Numeric types
  - Integers (with specified length in C, Fortran, or unspecified in others)
  - When unspecified, something that's ok in one system can be buggy in another
  - Some language support complex numbers (pair of reals)
  - Others offer it in a standard library (C++)
  - Some languages have a **rational number** type
  - Some languages have arbitrary precision numbers (Lisp and dialects of Scheme)
  - Fixed-point numbers (besides the usual floating-point)

## Types

- Enumeration types
  - Type weekday = (sun, mon, tue, wed, thu, fri, sat);
  - for today := mon to fri do begin ...

- Alternative to enumerations: just make them constants

- In C:

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};

typedef int weekday;
const weekday sun = 0, mon = 1, tue = 2,
              wed = 3, thu = 4, fri = 5, sat = 6;

enum mips_special_regs {gp = 28, fp = 30, sp = 29, ra = 31};
```

## Types

- Subranges

- Pascal

```
type test_score = 0..100;
     workday = mon..fri;
```

- Ada

```
type test_score is new integer range 0..100;
subtype workday is weekday range mon..fri;
```

  - This is a **derived** type
  - It is incompatible with integers

- **Space**: even if range is small, still 2 bytes

## Types

- Composite types
  - Records
  - Variant records (only one field is valid)
  - Arrays
  - Strings (arrays of characters)
  - Sets (powerset of base type)
  - Pointers (a **reference** to an object of the pointers base type)
  - Lists (no indexing, have to be traversed sequentially)
  - Files (notion of position, sequential reading in some cases)

# Types

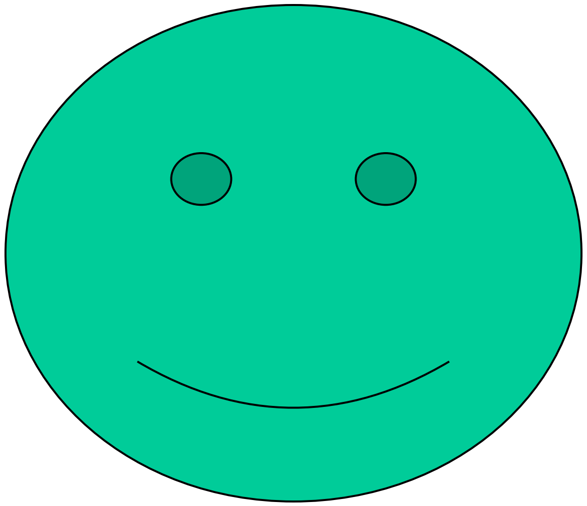
- Orthogonality
  - Making things be usable in any possible combination
  - For example expressions and statements are blurred in C and Algol
- Empty type (for orthogonality)
  - Allows to use functions as subroutines (only for their side-effects)

```
foo_index = insert_in_symbol_table(foo);
...
(void) insert_in_symbol_table(bar);     /* don't care where it went */
    /* cast is optional; implied if omitted */
```

## Types

- Aggregate assignment (for orthogonality) -- Ada

```
type person is record
          name : string (1..10);
          age : integer;
     end record;
p, q : person;
A, B : array (1..10) of integer;
...
p := ("Jane Doe  ", 37);
q := (age => 36, name => "John Doe  ");
A := (1, 0, 3, 0, 3, 0, 3, 0, 0, 0);
B := (1 => 1, 3 | 5 | 7 => 3, others => 0);
```



## Type checking

- Type equivalence
- Structural equivalence:
  - same components put together in the same way
  - Used (with some “wrinkles” in C and ML)
- Name equivalence
  - Based on lexical occurrence of definitions

## Type checking

- Structural equivalence specifics depend on language

```
type foo = record a, b : integer end;
```

```
type foo = record
    a, b : integer
end;
```

```
type foo = record
    a : integer;
    b : integer
end;
```

```
type foo = record
    b : integer;
    a : integer
end;
```

**?? Equivalent??**

## Type checking

- Structural equivalence specifics depend on language

```
type str = array [1..10] of char;

type str = array [1..2*5] of char;

type str = array [0..9] of char;
```

**?? Equivalent??**

- A compiler can determine structural equivalence by expanding their definitions recursively until all is left is a string of type constructors, field names and built-in types.

## Type checking

- Problems with structural equivalence
  - Types that are considered distinct by the programmer may be classified as equivalent because they happen to have the same structure

```
type student = record
        name, address : string
        age : integer

type school = record
        name, address : string
        age : integer

x : student;
y : school;
. . .
x := y;                  -- is this an error?
```

## Type checking

- Name definition
  - If the programmer wrote two definitions, then there is reason to call them different

- Possible problem with **aliases**

```
TYPE new_type = old_type;
```

```
TYPE stack_element = INTEGER;   (* or whatever type the user prefers *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
    ...
PROCEDURE push(elem : stack_element);
    ...
PROCEDURE pop() : stack_element;
    ...
```

**Aliases ok**

## Type checking

- Name definition
  - If the programmer wrote two definitions, then there is reason to call them different

- Possible problem with **aliases**

```
TYPE new_type = old_type;
```

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

**Aliases not ok**

## Type checking

- Possible problem with **aliases**
- Strict name equivalence (aliases are different types)
- Loose name equivalence (aliases are same types)
- Ada allows both, with derived types and subtypes

```
subtype stack_element is integer;
...
type celsius_temp is new integer;
type fahrenheit_temp is new integer;
```

- **Take a look at the example in the book**

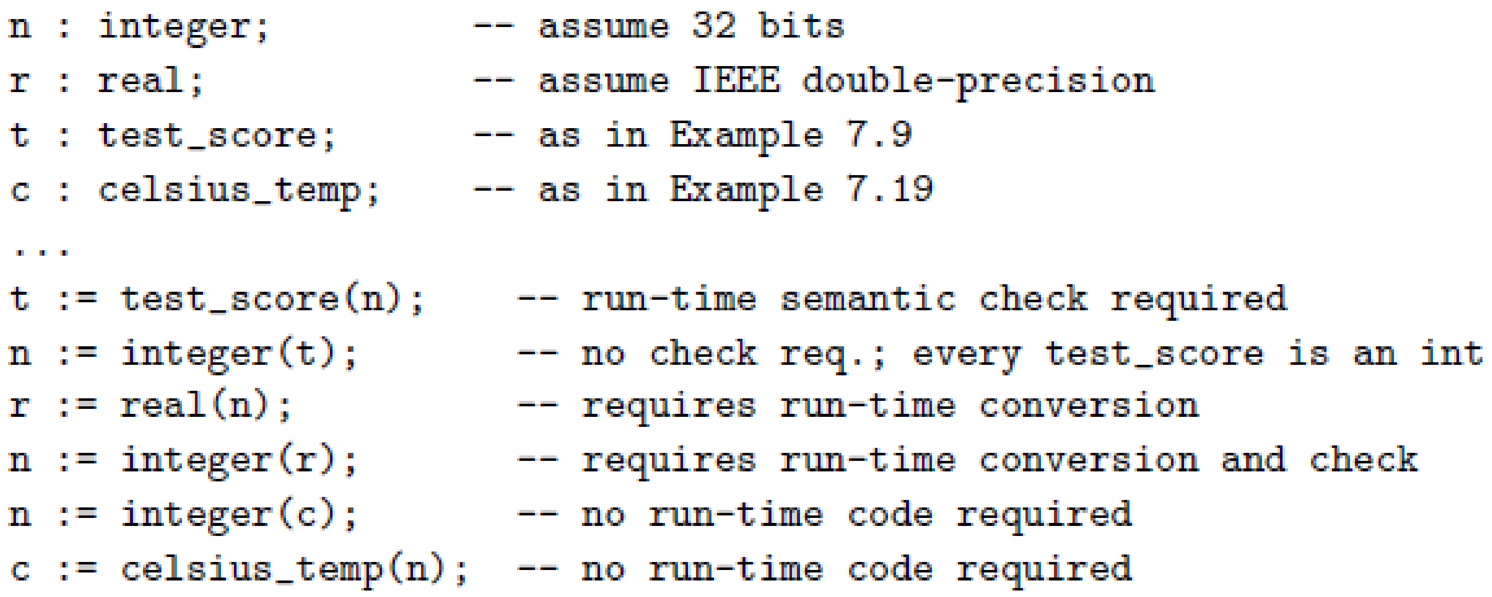
## Type checking

- Cast operations, aka type conversion

1. Language can use name equivalence but two types may be structurally equivalent. Then cast is conceptual and easy, so it uses no code.
2. Two types have different sets of values, but these intersect. Then code must be executed at run time to make sure that a value of type B that is used for something with type A, is also part of the values of A.
   - Dynamic semantic error
   - Check may be disabled, but with potential unsafety problems
3. Types have different low-level implementations, but still some correspondence. Different types of numbers, conversions is supported by machine instructions.

## Type checking

- An example



```
n : integer;          -- assume 32 bits
r : real;             -- assume IEEE double-precision
t : test_score;       -- as in Example 7.9
c : celsius_temp;     -- as in Example 7.19
...
t := test_score(n);     -- run-time semantic check required
n := integer(t);        -- no check req.; every test_score is an int
r := real(n);           -- requires run-time conversion
n := integer(r);        -- requires run-time conversion and check
n := integer(c);        -- no run-time code required
c := celsius_temp(n);   -- no run-time code required
```

# Type checking

- Non-converting type casts
  - Just interpret the bits of a variable as of being of different type

- Unchecked cast

```
-- assume 'float' has been declared to match IEEE single-precision
function cast_float_to_int is
    new unchecked_conversion(float, integer);
function cast_int_to_float is
    new unchecked_conversion(integer, float);
...
f := cast_int_to_float(n);
n := cast_float_to_int(f);
```

## Type checking

- **Coercion** is using some type where something else is expected
  - It's a form of implicit conversion
  - It often requires dynamic run-time checks
  - C and Fortran allow coercion to a great extent (unsafety)
  - See example in book

- Fortran 90 allows arrays and records to be intermixed if their types have the same shape.
- C doesn't allow records to be intermixed, and provides no operations that take whole arrays as operands (e.g. you can't assign an array)
- C++ allows user-definable coercions

## Type checking

- Constants are treated as special cases in the type checking rules
- Internally a compiler considers a constant to belong to a small number of types (like the *nil* constant)
  - Coercion may be used, but only for these constants if  not supported by language.

## Type checking

- **Generic reference types**. For writing general purpose container (collection) objects (lists, stacks, queues, sets etc) – system programming
  - void * (C,C++)
  - any (Clu)
  - address (Modula-2)
  - Etc

- Arbitrary values can be assigned into an object of a generic reference type
  - No concern about safety
- No operations with this object are allowed
- We can assign it back to a specific (normal) type, but we have to be careful with safety now (type casts, type tags in names)

## Type checking

- **Type inference**. Most times works as expected
- Subranges may be more tricky.

```
type Atype = 0..20;
     Btype = 10..20;
var  a : Atype;
     b : Btype;
```

a + b?

- Answer in Pascal: type of a+b is the base type (integer)

- In Ada: For loop index type is determined by the type of the loop bounds type
- If a result of an expression is assigned into an object of subarange type then dynamic checks may be needed. Or the compiler can try to keep track abou the possible min and max values of the expression.

## Type checking

- **Composite types and type inference**
- Most operators apply to built-in types
- There are exceptions

- Strings  (Ada)
- Sets (Pascal)

```
var  A : set of 1..10;
     B : set of 10..20;
     C : set of 1..15;
     i : 1..30;
...
C := A + B * [1..5, i];
```

## Records

- Records, structures, or classes where everything is public (C++,Java)
- Components are known as fields

```
type two_chars = packed array [1..2] of char;
    (* Packed arrays will be explained in Example 7.39.
       Packed arrays of char are compatible with quoted strings. *)
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```

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

# Records

- Nested records are allowed in most languages

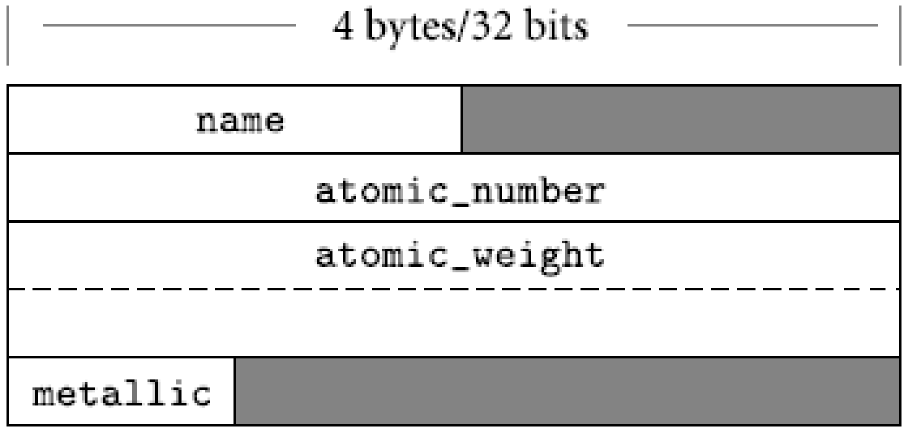
```
type short_string = packed array [1..30] of char;
type ore = record
    name : short_string;
    element_yielded : record
        name : two_chars;
        atomic_number : integer;
        atomic_weight : real;
        metallic : Boolean
    end
end;
```

```
type ore = record
    name : short_string;
    element_yielded : element
end;
```

- In some languages only the second alternative is allowed
- ML doesn’t care about order of fields

## Records

- Memory and records: Fields are usually kept close in memory

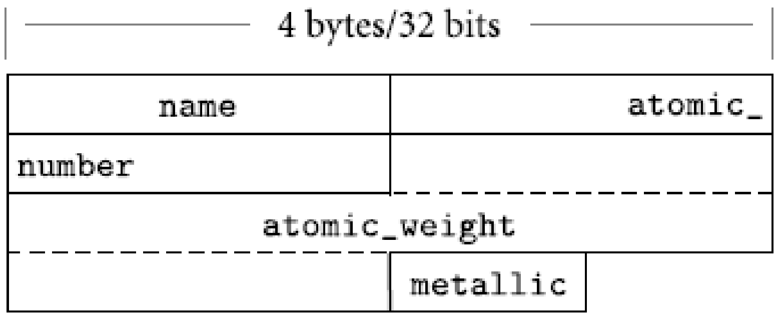


- As you can see, unused space may be a problem

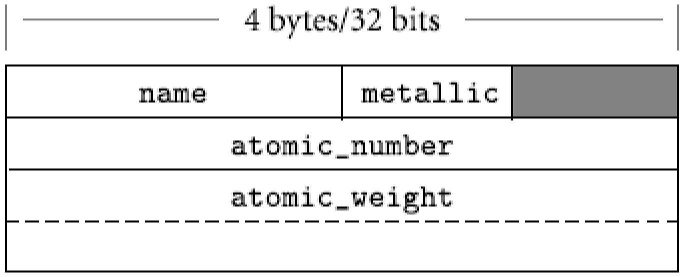
## Records

- In Pascal we can specify that a record is **packed**
- **Non-aligned** fields need to be read via multiple instructions

```
type element = packed record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean
end;
```

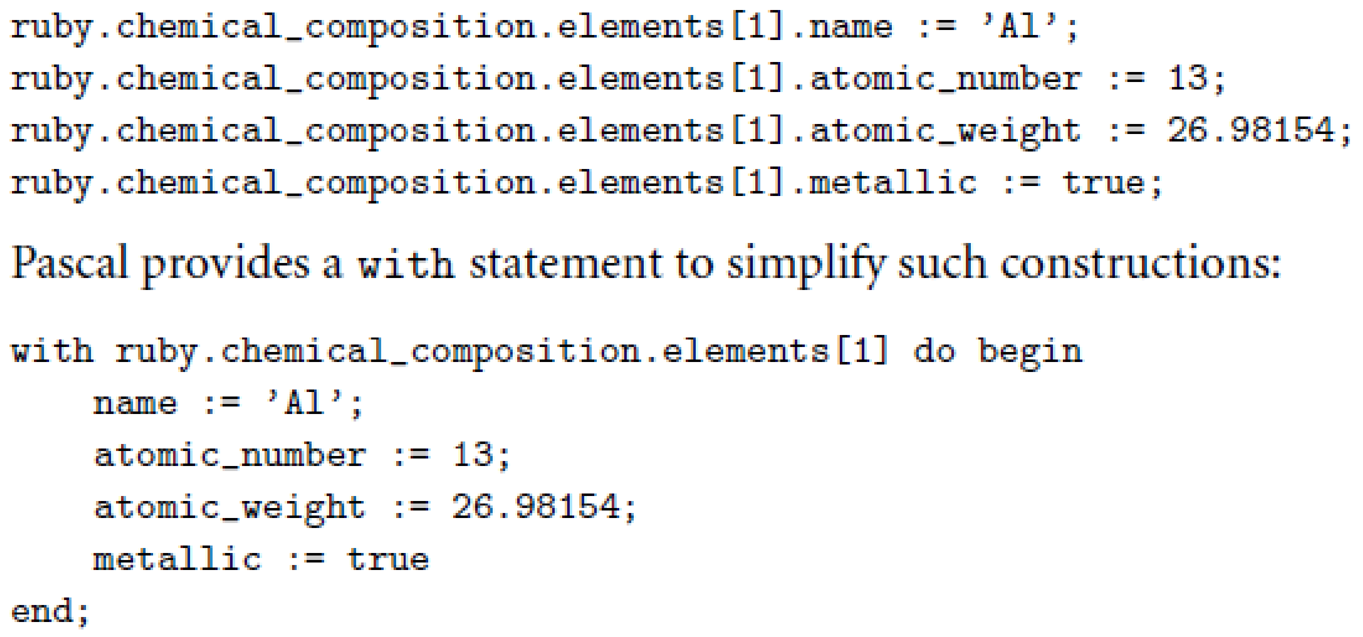


- Some compilers rearrange positioning in memory to minimize space loss
- Not always desirable



## Records

- **With** statements, allow handling deeply nested records



```
ruby.chemical_composition.elements[1].name := 'Al';
ruby.chemical_composition.elements[1].atomic_number := 13;
ruby.chemical_composition.elements[1].atomic_weight := 26.98154;
ruby.chemical_composition.elements[1].metallic := true;
```

Pascal provides a `with` statement to simplify such constructions:

```
with ruby.chemical_composition.elements[1] do begin
    name := 'Al';
    atomic_number := 13;
    atomic_weight := 26.98154;
    metallic := true
end;
```

## Records

- Variant records: One of some fields exists, the rest not, based on **tag, aka discriminant**

```
type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
            (* textual description of principal commercial source *)
        prevalence : real;
            (* fraction, by weight, of Earth's crust *)
      );
      false : (
        lifetime : real;
            (* half-life in seconds of the most stable known isotope *)
      )
end;
```

# Records

- Variant records: One of some fields exists, the rest not, based on **tag, aka discriminant**
- Coming from equivalence statements in Fortran
  - Saying that one of i,r,b is valid at any time

```
integer i
real r
logical b
equivalence (i, r, b)
```

- Safety problem:

```
r = 3.0
...
print '(I10)', i
```

## Records

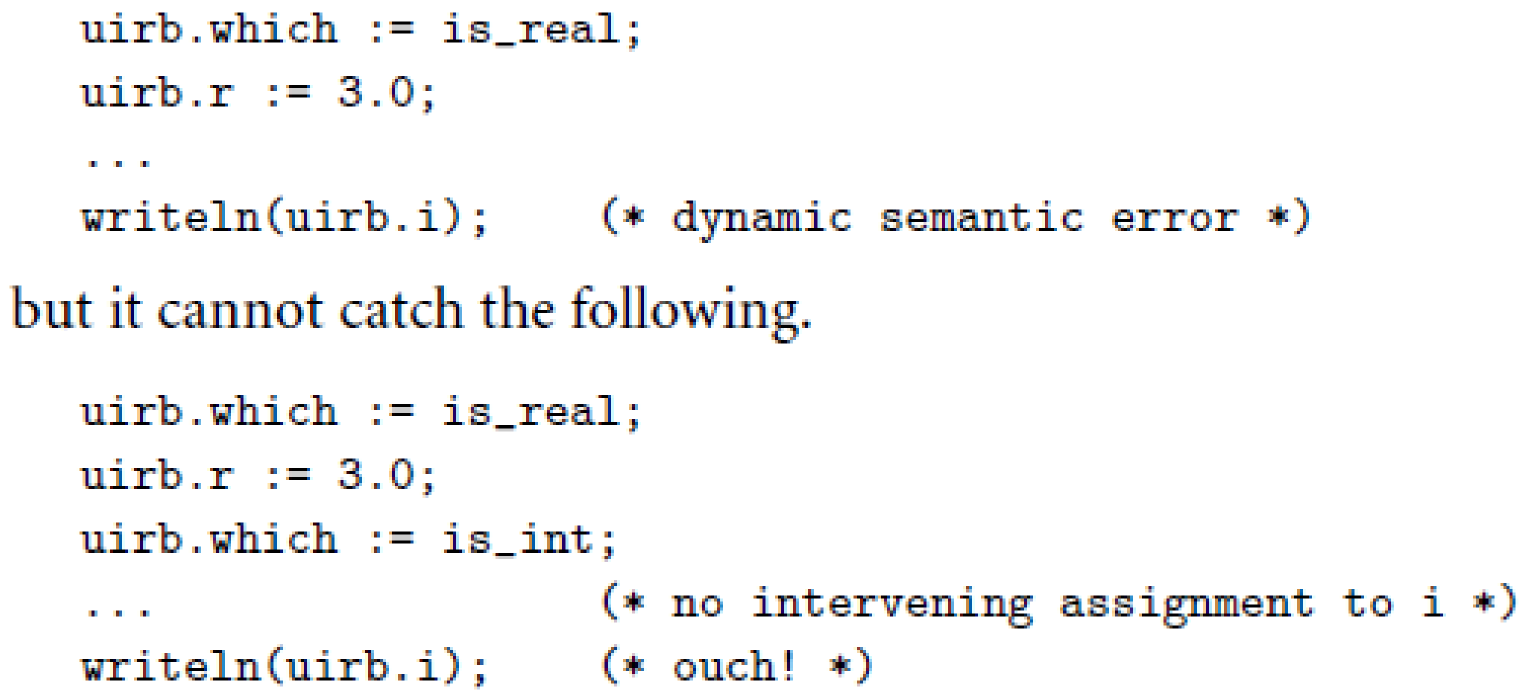
- Equivalence statement in Algol68
- Same name can have different types

```
union (int, real, bool) uirb
    # uirb can be an integer, a floating-point number, or a Boolean #
...
uirb := 1        # uirb is now an integer #
...
uirb := 3.14     # uirb is now a floating-point number #
```

- Needs extra bookeeping to remember current type

## Records

- Type safety problems with variant records



```
uirb.which := is_real;
uirb.r := 3.0;
...
writeln(uirb.i);    (* dynamic semantic error *)
```

but it cannot catch the following.

```
uirb.which := is_real;
uirb.r := 3.0;
uirb.which := is_int;
...                 (* no intervening assignment to i *)
writeln(uirb.i);    (* ouch! *)
```