Notes for CSC 2/454, Nov. 11 & 16, 2020

==========

A smorgasbord of types

scalar types -- one or two-dimensional
- discrete -- one-dimensional and countable
  - integer
  - boolean
  - char
  - enumeration
  - subrange
- rational
- real
- complex

composite types
- records/structs/tuples
- variants/unions
- arrays
  - strings
- sets
- pointers
- lists
- files

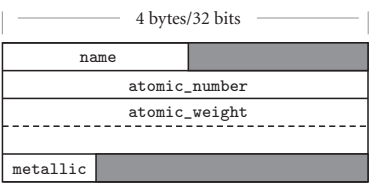
- mappings // common in scripting languages

Records
- usually laid out contiguously
- possible holes for alignment reasons
  - permits copying but ***not*** comparison with simple block operations

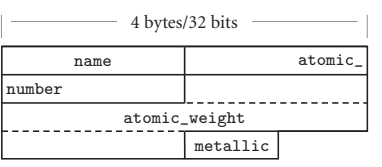
- example:

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

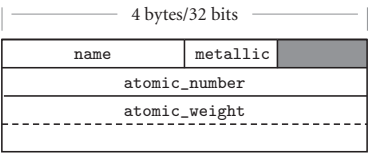
- layout on a 32-bit machine:



A few languages allow the programmer to specify that a record is ***packed***, meaning there are no (internal) holds, but fields may be unaligned.
- less space, but significant run-time access penalty



Smart compilers may re-arrange fields to minimize holes
- largest first or smallest first
  - latter maximizes # of fields with a small offset from the beginning



- C compilers promise not to rearrange

Unions (variant records)
- overlay space
- w/ tag: ***discriminated*** union
- w/out tag: nondiscriminated union
  - cause problems for type checking -- you don't know what is there
- ability to change tag and then access fields hardly any better
  - can make fields "uninitialized" when tag is changed (this generally requires extensive run-time support)
  - can require assignment of entire variant (w/ tag), as in Ada or OCaml

- Several languages (including Algol68, Ada, and ML) require access to variant portions of a record to be confined to a "conformity clause" (e.g., OCaml's match) that ensures type safety.

If structs and unions are independent, declarations can be quite ugly, as in this legacy C:

```
struct employee {
   ...
   union {
      struct { // hourly employee
         double hourly_pay;
         ...
      } S1;
      struct { // salaried employee
         double annual_salary;
         ...
      } S2;
   } U1;
};
...
this_employee.U1.S1.hourly_pay     // yuk!
```

Pascal unified records and variants:

```
type employee = record
   ...
   case boolean of    (* hourly? *)
     true:
      hourly_pay : real;
      ...
     false:
      annual_salary: real;
      ...
end;
...
this_employee.hourly_pay           // better
```

Recent versions of C and C++ achieve a similar effect with ***anonymous*** structs and unions. Strike out the S1, S2, U1 names above.

Note that the problem of uninitialized variables is more general than variant records. Some languages say variables start out with certain values (e.g. 0 for globals [but not locals!] in C). Many just say it's erroneous to use an uninitialized variable. A few actually try to prevent you from accessing one. In general, the only ways to do this are (1) restrict the language, e.g., as Java and C# do to ensure ***definite assignment***; (2) initialize variables automatically with a special "uninitialized" value and check most references at run time.