A smorgasbord of types

**Scalar types**
- One or two-dimensional
  - Integer
  - Character
  - Enumeration
  - Subrange
  - Rational
  - Real
  - Complex

**Composite types**
- Records/structs/tuples
- Variants/unions
- Arrays
- Strings
- Sets
- Pointers
- 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:**
```c
struct element {
  char name[2];
  int atomic_number;
  double atomic_weight;
  bool metallic;
}
```

**Layout on a 32-bit machine:**
```
<table>
<thead>
<tr>
<th>atomic_number</th>
<th>atomic_weight</th>
<th>metallic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>
```

- 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.

**Unions**
- Overlay space w/ tag:
  - Discriminated union
- Without 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:
```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:**
```pascal
type employee = record
  ...
  case boolean of
    true: hourly_pay: real;
    false: annual_salary: real;
    end;
  ...
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 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.