Records, structures

- Groupings of fields of various types
- Fields identified by name, not position
- Fields can generally be of any type (even other records)
- Declaration requires specification of fields, might require specification of field types
- Field access requires specification of the record variable and its field

Operations

- Assigning or copying field values
- Assigning or copying entire record (could be shallow or deep copy)
- Copying of compatible records may be possible (e.g.

  struct TypeA { int field1, field2, field3; };

  struct TypeB { int entry1, entry2; };

  TypeA x; TypeB y;

  What happens if we try x = y; or y = x;
- Changing structure of record (adding or removing fields)
- Get list of fields of a record (especially if fields can be added/removed)

Storage of records/structures

- As with arrays, are they stored on stack or in heap
- As with arrays, what happens if we resize (add new fields)
- Compiler replaces HLL field accesses with access to memory at a computed offset from the start of the record
- Memory alignment issues to consider within the record, e.g.

  struct MyData { char c; long i; char d; float f };

  If we store fields in sequence, we need to insert padding to adhere to memory alignment rules, or compiler may rearrange fields from largest to smallest

Passing or returning records

- Similar issues as when we considered passing or returning arrays (do we pass a full copy, or a reference?) with similar implications
- Languages do not necessarily apply the same rules to records as arrays (e.g. may pass address of arrays, but full copies of records)