Run-time storage layout:
focus on compilation, not interpretation
- Play how and where to keep data at run-time
- Representation of
  - int, bool, etc.
  - arrays, records, etc.
  - procedures
- Placement of
  - global variables
  - local variables
  - parameters
  - results

Data layout of scalars
Based on machine representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Use hardware representation (2, 4, or 8 bytes of memory, maybe aligned)</td>
</tr>
<tr>
<td>Bool</td>
<td>1 byte or word</td>
</tr>
<tr>
<td>Char</td>
<td>1-2 bytes or word</td>
</tr>
<tr>
<td>Pointer</td>
<td>Use hardware representation (2, 4, or 8 bytes, maybe two words if segmented machine)</td>
</tr>
</tbody>
</table>

Data layout of aggregates
- Aggregate scalars together
- Different compilers make different decisions
- The decisions are sometimes machine dependent
  - Note that through the discussion of the front-end, we essentially never mentioned the target machine
  - We didn’t in interpretation, either
  - But now it’s going to start to come up constantly

Layout of records
- Concatenate layout of fields
- Respect alignment restrictions
- Respect field order, if required by language
  - Why might a language choose to do this or not do this?

Layout of arrays
- Repeated layout of element type
- Respect alignment of element type
- How is the length of the array handled?
Layout of multi-dimensional arrays

- Recursively apply layout rule to subarray first
- This leads to row-major layout
- Alternative: column-major layout
- Most famous example: FORTRAN

Dynamically sized arrays

- Arrays whose length is determined at run-time
- Different values of the same array type can have different lengths
- Can store length implicitly in array
- Where? How much space?
- Dynamically sized arrays require pointer indirection
  - Each variable must have fixed, statically known size

Dope vectors

- PL/0 handled arrays differently, in particular storage of the length
- It used something called a dope vector, which was a record consisting of
  - A pointer to the array
  - The length of the array
- Arrays could change locations in memory and size quite easily

String representation

- A string is an array of characters
  - So, can use array layout rule for strings
- Pascal: strings have statically determined length
  - Layout like array with statically determined length
- Other languages: strings have dynamically determined length
  - Layout like array with dynamically determined length
  - Alternative: use special end-of-string character (e.g., \0)

Storage allocation strategies

- Given layout of data structure, where in memory to allocate space for each variable/data structure?
- Key issue: what is the lifetime (dynamic extent) of a variable/data structure?
  - Whole execution of program (e.g., global variables) ⇒ Static allocation
  - Execution of a procedure activation (e.g., locals) ⇒ Stack allocation
  - Variable (dynamically allocated data) ⇒ Heap allocation

Parts of run-time memory

- Code/Read-only data area
  - Shared across processes running same program
- Static data area
  - Can start out initialized or zeroed
- Heap
  - Can expand upwards through (e.g., system call)
- Stack
  - Expands/contracts downwards automatically
Static allocation
- Specially allocate variables/data structures with global lifetime
  - Global variables
  - Compile-time constant strings, arrays, etc.
  - `static` local variables in C, all locals in FORTRAN
  - Machine code
- Compiler uses symbolic addresses
- Linker assigns exact address, patches compiled code

Stack allocation
- Stack-allocate variables/data structures with LIFO lifetime
  - Data doesn’t outlive previously allocated data on the same stack
  - Procedure activation records allocated on a stack
    - A stack-allocated activation record called a stack frame
    - Frame includes formals, locals, static link of procedure
    - Dynamic link points to stack frame above
  - Fast to allocate and deallocate storage
  - Good memory locality

Constraints on stack allocation
- Stack allocation required no references to stack-allocated data after returns
- This is violated by general first-class functions

Heap allocation
- For data with unknown lifetime
  - `new/malloc` to allocate space
  - `delete/free`/garbage collection to deallocate space
  - Heap-allocate activation records of first-class functions
  - Relatively expensive to manage
  - Can have dangling reference, storage leaks
    - Garbage collection reduces (but may not eliminate) these classes of errors

Stack frame layout
- Need space for
  - Formals
  - Locals
  - Dynamic link
  - Static link
  - Other run-time data (e.g., return address, saved registers)
- Assign dedicated registers to support access to stack frames
  - Frame pointer (FP): ptr to beginning of stack frame (fixed)
  - Stack pointer (SP): ptr to end of stack (can move)
Key property

- All data in stack frame is at a fixed, statically computed offset from the FP
- This makes it easy to generate fast code to access the data in the stack frame
  - And even lexically enclosing stack frames
- Can compute these offsets solely from the symbol tables
  - Based also on the chosen layout approach