Run-time storage layout:  
*focus on compilation, not interpretation*

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

### Data layout of aggregates
- Aggregate scalars together
- Different compilers make different decisions
- Decisions are sometimes machine dependent
  - Note that through the discussion of the front-end, we never mentioned the target machine
  - We didn’t in interpretation, either
  - But now it’s going to start to come up constantly
  - Necessarily, some of what we will say will be “typical”, not universal.

### 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?
- Respect contiguity?

```plaintext
z : record
  b : bool;
  i : int;
  m : record
    b : bool;
    c : char;
  end
  j : int;
end;
```

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

```plaintext
x : array [5] of record
  i : int;
  c : char;
end;
```
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/1 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
  - Subscript bounds for each dimension
- Arrays could change locations in memory and size quite easily

String representation

- A string – an array of characters
- So, can use array layout rule for strings
- Pascal, C strings: 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: special end-of-string char (e.g., \0)

Storage allocation strategies

- Given layout of data structure, where in memory to allocate space for each instance?
- 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., sbrk) system call
- Stack
  - Expands/contracts downwards automatically
### Static allocation

- Statically allocate variables/data structures with global lifetime
- Machine code
- Compile-time constant scalars, strings, arrays, etc.
- Global variables
- Static locals in C, all variables in FORTRAN
- 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
- Stack-allocate procedure activation records
  - A stack-allocated activation record = a stack frame
  - Frame includes formals, locals, temps
  - And housekeeping: static link, dynamic link, ...
- Fast to allocate and deallocate storage
- Good memory locality

### Stack allocation II

- What about variables local to nested scopes within one procedure?

```plaintext
procedure P(){
    int x;
    for(int i=0; i<10; i++){
        double x;
        ...
    }
    for(int j=0; j<10; j++){
        double y;
        ...
    }
}
```

### Constraints on stack allocation

- No references to stack-allocated data allowed after returns
- This is violated by general first-class functions

```plaintext
proc foo(x:int): *int;
proc bar(y:int):int
begin
    return x + y;
end bar;

begin
    return bar;
end foo;

var f: proc type(int): int;
var g: proc type(int): int;
f := foo(3); g := foo(4);
output := *f(5); output := *g(6);
```

### Constraints on stack allocation

- Also violated if pointers to locals are allowed

```plaintext
proc foo [x:int]: *int;
var y:int;
begin
    y := x + 2;
    return 4y;
end foo;

var w,z:*int;
x := foo(3);
w := foo(4);
output := *z;
output := *w;
```

### Heap allocation

- For data with unknown lifetime
  - new/malloc to allocate space
  - delete/free/garbage collection to deallocate
- 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
  - Various housekeeping data
    - Dynamic link (pointer to caller's stack frame)
    - Static link (pointer to lexically enclosing stack frame)
    - Return address, saved registers, ...
- Dedicate registers to support stack access
  - FP - frame pointer: ptr to start of stack frame (fixed)
  - SP - stack pointer: 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

Stack Layout

Accessing locals

- If a local is in the same stack frame then
  \( t := *(fp + local\_offset) \)
- If in lexically-enclosing stack frame
  \( t := *(fp + static\_link\_offset) \)
  \( t := *(t + local\_offset) \)
- If farther away
  \( t := *(fp + static\_link\_offset) \)
  \( t := *(t + static\_link\_offset) \)
  \( \ldots \)
  \( t := *(t + local\_offset) \)

At compile-time…

- …need to calculate
  - Difference in nesting depth of use and definition
  - Offset of local in defining stack frame

Calling conventions

- Define responsibilities of caller and callee
  - To make sure the stack frame is properly set up and torn down
  - Some things can only be done by the caller
  - Other things can only be done by the callee
  - Some can be done by either
  - So, we need a protocol
### PL/0 calling sequence
- **Caller**
  - Evaluate actual args
  - Order?
  - Alternative: First k args in registers
  - Push callee's static link
  - Or in register? Before or after stack arguments?
  - Execute call instruction
  - Hardware puts return address in a register
- **Callee**
  - Save return address on stack
  - Save callee's frame pointer (dynamic link) on stack
  - Save any other registers that might be needed by caller
  - Allocates space for locals, other data
  - Locals stored in what order?
  - Set up new frame pointer (fp := sp)
  - Start executing callee's code

### PL/0 return sequence
- **Callee**
  - Deallocate space for local, other data
  - sp := sp + size_of_locals + other_data
  - Restore caller's frame pointer, return address & other regs, all without losing addresses of stuff still needed in stack
  - Execute return instruction
- **Caller**
  - Deallocate space for callee's static link, args
  - sp := fp
  - Continue execution in caller after call

### Accessing callee procedures similar to accessing locals
- Call to procedure declared in same scope:
  ```
  static_link := fp
call p
  ```
- Call to procedure in lexically-enclosing scope:
  ```
  static_link := *(fp + static_link_offset)
call p
  ```
- If farther away
  ```
  t := *(fp + static_link_offset)
t := *(t + static_link_offset)
... static_link := *(t + static_link_offset)
call p
  ```

### Some questions
- Return values?
- Local, variable-sized, arrays
  ```
  proc P(int n) {
  var x array[1 .. n] of int;
  var y array[-5 .. 2*n] of array[1 .. n] int;
  ...
  }
  ```
  - Max length of dynamic-link chain?
  - Max length of static-link chain?

### Exercise: apply to this example
```plaintext
module M:
  var x:int;
  proc P(y:int);
  proc Q(y:int);
  begin R(x+y); end Q;
  proc R(z:int);
  begin P(x+y+z); end R;
  begin Q(x+y); end P;
begin
  x := 1;
P(2);
end M.
```

### What do these mean?
```plaintext
proc P(int a);
begin
  output := a;
  output := a+1;
  a := a+1;
  output := a;
end;

int i=2;
P(i); output i;
P(2); output 2;

proc Q(int a,int b);
begin
  c := a;
  a := b;
b := c;
end;

int i=2; j=3;
P(i); Q(i,j);
```
Parameter passing

- When passing args, need to support right semantics
- Issue #1: when is argument expression evaluated?
  - Before call?
  - If and when needed by callee?
- Issue #2: what happens if callee assigns to formal?
  - Is this visible to the caller? If so, when?
  - What happens with aliasing among arguments and lexically visible variables?
- Different choices lead to
  - different representations for passed arguments and
  - different code to access formals

Parameter passing modes

- call-by-value
- call-by-sharing
- call-by-reference
- call-by-value-result
- call-by-name
- call-by-need
- ...

Call-by-value

- Assignment to formal doesn’t affect caller’s value
- Implementation: pass copy of argument value
- Trivial for scalars
- Inefficient for aggregates

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```

Call-by-reference

- Assignment to formal changes actual value in caller
- Immediately
- Actual must be value
- Implementation: pass pointer to actual
- Efficient for big data structures
- References to formal must do extra dereference

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```

Big immutable data

- for example, a constant string
- Suppose language has call-by-value semantics
- But, it’s expensive to pass by-value
- Could implement as call-by-reference
  - Since you can’t assign to the data, you don’t care
  - Let the compiler decide?

Call-by-value-result

- Assignment to formal copies final value back to caller on return
- “copy-in, copy-out”
- Implement as call-by-value with copy back when procedure returns
- More efficient than call-by-reference
  - For scalars?
  - For arrays?

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```
Call-by-result

```plaintext
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```

Ada: in, out, in out

- Programmer selects intent
- Compiler decides which mechanism is more efficient
- Program's meaning "shouldn't" depend on which is chosen

---

Call-by-name, call-by-need

- Variations on lazy evaluation
  - Only evaluate argument expression if and when needed by callee
- Supports very cool programming tricks
  - Somewhat hard to implement efficiently in traditional compilers
  - Thunks
- Largely incompatible with side-effects
  - So more common in purely functional languages like Haskell and Miranda
  - But did appear first in Algol-60

---

Call-by-name

- Replace each use of a parameter in the callee, by the text of the actual parameter, but in the caller's context
- And evaluation of the actual might return different values each time

---

Jensen's device

- How to implement the equivalent of a math formula like $\sum_{i=0}^{n} A[2i]$?
- Pass by-reference or by-value do not work, since they can only pass one element of $A$
- So: Jensen's device

---

A classic problem:
a procedure to swap two elements

```plaintext
int proc sum(j,lo,hi,Aj);
int j, lo, hi, Aj, s;
begin
  s := 0;
  for j := lo to hi do
    s := s + Aj;
  end;
  return s;
end;
```

```plaintext
proc square(x);
  int x;
  begin
    x := x * x
  end;
end;
```

```plaintext
proc swap(x, y);
  int temp;
  begin
    temp := a;
    a := b;
    b := temp;
  end;
end;
```

```plaintext
int x, y;
  x := 2;
y := 5;
  swap(x, y);
```

```plaintext
int j, z[10];
j := 2;
z[2] := 5;
  swap(j, z[j]);
```
Call-by-name advantages

- Textual substitution is a simple, clear semantic model
- There are some useful applications, like Jensen's device
- Argument expressions are evaluated lazily

Call-by-name disadvantages

- Repeatedly evaluating arguments can be inefficient
- Pass-by-name precludes some standard procedures from being implemented
- Pass-by-name is difficult to implement

thunks

- Call-by-name arguments are compiled to thunks, special parameter-less procedures
  - One gives value of actual, appropriately evaluated in caller's environment
  - Other gives l-value, again in caller's environment
- Thunks are passed into the called procedure and called to evaluate the argument whenever necessary

Parameters and compiling

- There is an intimate link between the semantics of a programming language and the mechanisms used for parameter passing
- Maybe more than other programming language constructs, the connection is extremely strong between implementation and language semantics in this area

PL/0 storage allocation

- How and when it is decided how big a stack frame will be?
  - It's necessary that the frame always be the same size for every invocation of a given procedure
- Also, how and when is it decided exactly where in a stack frame specific data will be?
  - Some pieces are decided a priori (such as the return address)
  - Others must be decided during compile-time, such as local variables (since the number and size can't be known beforehand)
- This is all done during the storage allocation phase

PL/0 storage allocation

```c
void SymTabScope::allocateSpace() {
    _localsSize = 0;
    _formalsSize = 0;
    for (int i = 0; i < _symbols->length(); i++)
    {
        _symbols->fetch(i)->allocateSpace(this);
    }
    for (int j = 0; j < _children->length(); j++)
    {
        _children->fetch(j)->allocateSpace();
    }
}
```
```c++
int SymTabScope::allocateFormal(int size) {
    int offset = _formalsSize;
    _formalsSize += size;
    return offset;
}

int SymTabScope::allocateLocal(int size) {
    int offset = _localsSize;
    _localsSize += size;
    return offset;
}

void VarSTE::allocateSpace(SymTabScope* s) {
    int size = _type->size();
    _offset = s->allocateLocal(size);
}
```