Run-Time Storage Layout

Plan how & where to keep data at run-time

Representation of

- int, bool, ...
- arrays, records, ...
- procedures

Placement of

- global variables
- local variables
- formal parameters
- results

Data layout

Determined by type of data

- scalar data based on machine representation
- aggregates group these together

Integer: use hardware representation
(2,4, and/or 8 bytes of memory, maybe aligned)

Bool: e.g. 0 or 1, 1 byte or word

Char: 1-2 bytes or word

Pointer: use hardware representation
(2,4, or 8 bytes, maybe two words if segmented machine)

Layout of records

Concatenate layout of fields

- respect alignment restrictions
- respect field order, if required by language

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


Layout of arrays

Repeat layout of element type

- respect alignment of element type

```
a: array [5] of
        record
          i: int;
          c: char;
         end;
```

Array length?

Multi-dimensional arrays

Recursively apply layout rule to subarray first

```
a: array [3] of
        array [5] of
             record
               i: int;
               c: char;
             end;
```

Leads to **row-major** layout

Alternative: **column-major**

Dynamically sized arrays

Array whose length is determined at run-time

- different values of same array type can have different lengths

```
a: array of
        record
          i: int;
          c: char;
         end;
```

Store length implicitly in array

- where?
- how much space?

Dynamically sized arrays require pointer indirection

- each variable must have fixed, statically known size

String representation

String ≈ array of chars

- can use array layout rule to layout 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)

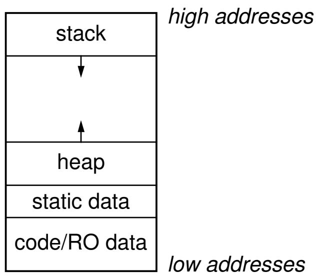
Storage allocation strategies

Given layout of data structure,
where 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 (global variables)
  ⇒ **static** allocation
- execution of a procedure activation (formals, local vars)
  ⇒ **stack** allocation
- variable (dynamically-allocated data)
  ⇒ **heap** allocation

Parts of run-time memory



Code/RO data area
- read-only data & machine instruction area
- shared across processes running same program

Static data area
- place for read/write variables at fixed location in memory
- can start out initialized, or zeroed

Heap
- place for dynamically allocation/freed data
- can expand upwards through `sbrk` system call

Stack
- place for stack-allocated/freed data
- expands/contracts downwards automatically

Static allocation

Statically 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 address

Linker assigns exact address, patches compiled code

Stack allocation

Stack-allocate variables/data structures with **LIFO** lifetime
- last-in first-out (stack discipline):
  data structure doesn't outlive previously allocated data structures on same stack

Procedure activation records usually allocated on a stack
- a stack-allocated a.r. called a **stack frame**
- frame includes formals, locals, static link of procedure
- dynamic link = stack frame above

Fast to allocate & deallocate storage

Good memory locality

Problems with stack allocation

Stack allocation works only when can't have references to stack allocated data after data returns

Violated if first-class functions allowed

```
procedure foo(x:int):proctype(int):int;
  procedure bar(y:int):int;
  begin
    return x + y;
  end bar;
begin
  return bar;
end foo;

var f:proctype(int):int;
var g:proctype(int):int;

f := foo(3);
g := foo(4);

output := f(5);
output := g(6);
```

Violated if pointers to locals allowed

```
procedure foo(x:int):*int;
   var y:int;
begin
   y := x * 2;
   return &y;
end foo;

var z:*int;
var w:*int;

z := foo(3);
w := foo(4);

output := *z;
output := *w;
```


Heap allocation

Heap-allocate variables/data structures with unknown lifetime

- `new`/`malloc` to allocate space
- `delete`/`free`/garbage collection to deallocate space

Heap-allocate activation records (environments at least) of first-class functions

Relatively expensive to manage

Can have dangling references, storage leaks if don't `free` right

- use automatic garbage collection in place of manual `free` to avoid these problems

Stack frame layout

Need space for:

- formals
- local variables
- dynamic link (ptr to calling stack frame)
- static link (ptr to lexically-enclosing stack frame)
- other run-time data (e.g. return addr, saved registers)

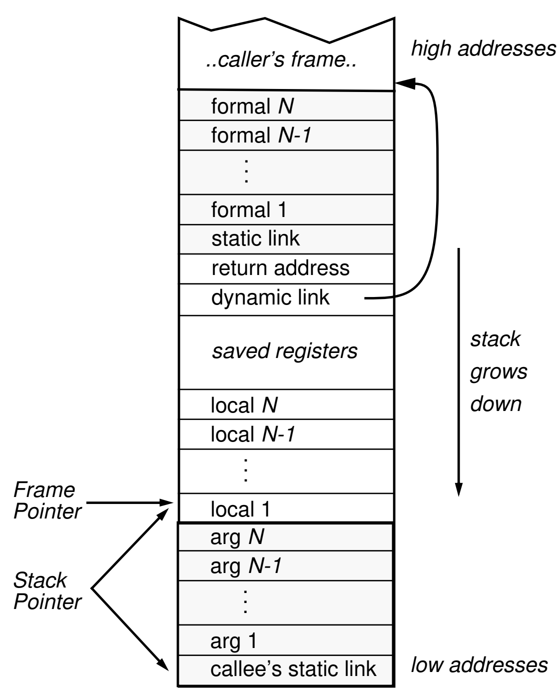
Assign dedicated register(s) to support access to stack frames

- 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 **fixed, statically computed** offset from FP

- easy to generate fast code to access data in stack frame, even lexically enclosing stack frames
- compute all offsets solely from symbol tables

PL/0 stack frame layout



Calling conventions

Need to define responsibilities of caller and callee
in setting up, tearing down stack frame

Only caller can do some things
Only callee can do other things
Some things could be done by both

Need a protocol

PL/0 calling sequence

Caller:

- evaluates actual arguments, pushes them on stack
  - in what order?
  - alternative: 1st *k* arguments in registers
- pushes static link of callee on stack
  - or in register? before or after stack arguments?
- executes call instruction
  - return address stored in register by hardware

Callee:

- saves return address on stack
- saves caller's frame pointer (dynamic link) on stack
- saves any other registers that might be needed by caller
- allocates space for locals, other data
  e.g. `sp := sp - size_of_locals - other_data`
  - locals stored in what order?
- sets up new frame pointer
  e.g. `fp := sp`
- starts running callee's code...

PL/0 return sequence

Callee:

- deallocates space for locals, other data
  e.g. `sp := sp + size_of_locals + other_data`
- restores caller's frame pointer from stack
- restores return address from stack
- executes return instruction

Caller:

- deallocates space for callee's static link, args
  e.g. `sp := fp`
- continues execution in caller after call...

PL/0 storage allocation

```
void SymTabScope::allocateSpace() {
   foreach sym
      sym->allocateSpace(s);
   foreach child scope
      child->allocateSpace();
}

void STE::allocateSpace(SymTabScope* s) {
   // no space for constants, procedures
}

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

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

// similar code for allocating Formals
```

Accessing locals

If in same stack frame:

```
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)
...
t := *(t + static_link_offset)
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

Accessing callee procedure static links

Similar to accessing locals; just stop before last indirection

If calling procedure declared in same stack frame:

```
static_link := fp
call p
```

If calling procedure declared in lexically-enclosing stack frame:

```
static_link := *(fp + static_link_offset)
call p
```

If calling procedure declared farther away:

```
t := *(fp + static_link_offset)
t := *(t + static_link_offset)
...
static_link := *(t + static_link_offset)
call p
```

At compile-time, need to calculate:

- difference in nesting depth of caller and callee

Example

```
module M;
   var x:int;
   procedure P(y:int);
      procedure Q(y:int);
         begin R(x+y); end Q;
      procedure R(z:int);
         begin P(x+y+z); end R;
   begin Q(x+y); end P;
begin
   x := 1;
   P(2);
end M.
```


Parameter passing

When passing arguments, need to support right semantics

An issue: when is argument expression evaluated?

- before call, or if & when needed by callee?

Another issue: what happens if formal assigned in callee?

- effect visible to caller? if so, when?
- what effect in face of aliasing among
  arguments, lexically visible variables

Different choices lead to different representations
for passed arguments and different code to access formals

Some parameter passing modes

Parameter passing options:

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

Call-by-value

If formal is assigned, doesn't affect caller's value

```
var a:int;
procedure foo(x:int, y:int);
begin
   x := x + 1; y := y + a;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement by passing copy of argument value

- trivial for scalars: ints, bools, etc.
- inefficient for aggregates: arrays, records, strings, ...

Call-by-sharing

If implicitly reference aggregate data via pointer
(e.g. Java, Lisp, Smalltalk, ML, ...)
then call-by-sharing is call-by-value applied to implicit pointer

- "call-by-pointer-value"
- efficient, even for big aggregates
- assignments of formal to a different aggregate (e.g. `f := x`) don't affect caller
- updates to contents of aggregate (e.g. `f[i] := x`) visible to caller immediately
- aliasing/sharing relationships preserved

Call-by-reference

If formal is assigned, actual value is changed in caller

- change occurs immediately
- assumes actual is an lvalue

```
var a:int;
procedure foo(var x:int, var y:int);
begin
   x := x + 1; y := y + a;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement by passing pointer to actual

- efficient for big data structures
- references to formal must do extra dereference!

If passing big immutable data (e.g. constant string) by value, can implement as call-by-reference

- can't assign to data, so can't tell it's a pointer

Call-by-value-result

If formal is assigned,
final value copied back to caller when callee returns

- "copy-in, copy-out"

```
var a:int;
procedure foo(in out x:int, in out y:int);
begin
   x := x + 1; y := y + a;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement as call-by-value, with assignment back when procedure returns

- more efficient for scalars than call-by-reference

Ada: `in out` either call-by-reference or call-by-value-result

- compiler can decide which is more efficient
- program "shouldn't" depend on which is chosen

Call-by-result

Formals assigned to return extra results;
no incoming actual value expected

- "out parameters"

```
var a:int;
procedure foo(out x:int, out y:int);
begin
   x := 1; y := a + 1;
end foo;
a := 2;
foo(a, a);
output := a;
```

Implement as in call-by-reference or call-by-value-result, depending on desired semantics

Call-by-name, call-by-need

Variations on lazy evaluation

- only evaluate argument expression if & when needed by callee function

Supports very cool programming tricks

Hard to implement efficiently in traditional compiler

Incompatible with side-effects
⇒ only in purely functional languages, e.g. Haskell, Miranda