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
- arguments
- results

Where data should be placed
- static, global memory
- stack memory
- heap memory

Data layout

Determined by type of data
- scalar data based on machine representation
- aggregates group these together

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

Layout of records

Concatenate layout of fields, respecting alignment restrictions

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

Layout of arrays

Repeat layout of element type, respecting alignment

```
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**

String representation

String ≈ array of chars
- can use array layout rule to layout strings

How to determine length of string at run-time?
- Pascal: strings have statically-determined length
- C: special terminating character
- High-level languages: explicit length field

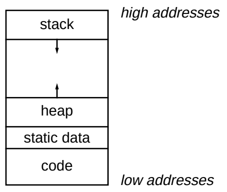
Storage allocation strategies

Given layout of data structure,
where to allocate space for each variable/data structure?

Key issue: what is the **lifetime** of a variable/data structure?

- whole execution of program (global variables)
  ⇒ **static** allocation
- execution of a procedure activation (arguments, local variables)
  ⇒ **stack** allocation
- variable (dynamically-allocated data)
  ⇒ **heap** allocation

Parts of run-time memory (UNIX)



Code area

- read-only machine instruction area
- shared across processes running same program

Static data area

- place for read/write variables at fixed location in memory
- can be initialized, or cleared

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 (text segment)

Compiler uses symbolic address
Linker determines exact address

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

- stack-allocated activation record called a **stack frame**
- frame includes:
  - arguments not passed in registers
  - locals
  - temporary values
  - machine state
  - static ("access") link to stack frame of lexically enclosing procedure
  - dynamic ("control") link = pointer to calling stack frame

Fast to allocate & deallocate storage (just change the SP)
Good memory locality
Good match with procedure calling conventions

Problems with stack allocation

Stack allocation works only when can't have references to stack-allocated data after procedure 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 -> **dangling reference**

```
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

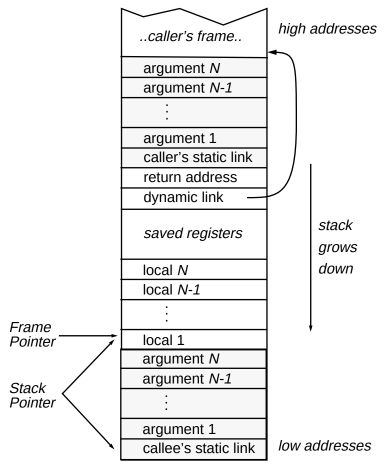
Stack frame layout

Need space for:
- arguments not passed in registers
- local variables
- temporary values
- dynamic link (ptr to calling stack frame)
- static link (ptr to lexically-enclosing stack frame)
- machine state (saved registers, possibly return address)

Assign dedicated register(s) to support access to stack frames
- stack pointer (SP): ptr to end of stack

or
- frame pointer (FP): ptr to beginning of stack frame (fixed)
- stack pointer (SP): ptr to end of stack (can move)

PL/0 stack frame layout



Calling conventions

Composition of the stack frame:
- how parameters are passed
- where return value is saved
- what else is on the stack

Separation of responsibilities between 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

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
  - 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 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 code...

PL/0 return sequence

Callee:
- deallocates space for locals, other data
  e.g. `sp := sp + size_of_locals + other_data`
- restores callee saved registers
- 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...

Static linkage

Need to connect stack frame to
stack frame holding values of lexically-enclosing variables

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

Static linkage

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

PL/0 storage allocation

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


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


int SymTabScope::allocateLocal(int size) {
   int offset = _localsSize;
   _localsSize += size;  // FIX THIS!
   return offset;
}
int SymTabScope::allocateFormal() { similar }
```

Example

proc foo (f1 : int, f2 : array[5] of bool);
  var l1 : array[3] of int;
  var l2 : bool;
  var l3 : int;
begin ... end foo

...			
proc	“foo”	int, array [5] of bool:void	...
...			

			offset	a formal?
var	“f1”	integer	0	yes
var	“f2”	array[5] of bool	4	yes
var	“l1”	array[3] of int	0	no
var	“l2”	bool	12	no
var	“l3”	int	16	no

Parameter passing

When passing arguments, need to support right semantics

- values
- locations
- text of parameter

Also want an efficient representation

==> lead to different representations for passed arguments
and different code to access formals

Parameter passing semantics:

- call-by-value
- call-by-reference
- call-by-value-result
- call-by-result
- ...

Call-by-value

Parameters are evaluated & r-values passed to procedure

- 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 ints, bools, etc.
- inefficient for arrays, records, strings, ...

Used in C, Pascal

Call-by-reference

Parameters are evaluated & l-values passed to procedure
- if formal is assigned, actual value is changed in caller
- change occurs immediately

```
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

Used in some Fortran implementations

How C emulates call by reference

```
foo (x,y)
int *x, *y;
   {*x = *x + 1;
   *y = *y + 1;
   }
main ()
{  int a;
   a = 2;
   foo (&a, &a);
   printf (“%d\n”, a);
}
```

Call-by-value-result

Arguments first passed as r-values
- If formal is assigned,
  final value copied back to caller **when callee returns**
- “copy-in, copy-out”, “copy-restore” (book)

```
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
- programmer should not rely on which it is

Used in some Fortran implementations

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

Used by Ada