Where to Store Variables?

**Static Allocation**
Variables created at compile-time
Size and address known at compile-time

**Stack Allocation**
Variables placed in activation records on a stack
Variables are created / destroyed in LIFO order

**Heap Allocation**
Size and address determined at run-time
Creation / destruction of data occurs in any order

---

Static Allocation

**Early Languages (FORTRAN)**
Each variable is placed in memory (“static allocation”)
Fortran had routines, but...
  • No stack
  • Recursion was not possible
Values of a routine’s variables are retained across invocations
  Initialization vs. re-initialization
Each variable’s size must be known at compile-time
  Dynamic arrays?
Stack Allocation

Each variable is "local" to some routine

Invoke a routine?
Allocate storage for its variables
(and initialize it?)

Return?
Pop frame
(Variables are destroyed)

Consider one routine (e.g., “quicksort”)
Many activations, many frames
⇒ Many copies of each local variable
Local variables:
Each invocation has its own set of variables
The “currently active” invocation
Its variables will be in the frame on top of stack.
Every reference to a local variable...
will access data in the top frame

General Idea

In the SPARC

References to a local variable in the currently active routine...

*(StackTop + offset_{X})  ld [%fp-48],%l5
Laying Out the Frame

Each local (and temp) variable has a size

“C” int: 4 bytes,
    double: 8 bytes, ....

Each local and temp variable needs an “offset”

```
for each procedure (or block) do
    offset = 0;
    for each local and temp variable do
        assign this variable to current offset
        offset = offset + this variable’s length
    endFor
endFor
```

May start at some other value (-4)
We’ll use -4
Laying Out the Frame

Each local (and temp) variable has a size

“C” int: 4 bytes,
    double: 8 bytes, ...
“PCAT” all variables: 4 bytes

Each local and temp variable needs an “offset”

```
for each procedure (or block) do
    offset = 0;
    for each local and temp variable do
        assign this variable to current offset
        offset = offset + this variable’s length
    endFor
endFor
```

We’ll treat “main” body as just another routine.
It will have a frame

Global variables
Treat identically to local variables for procedures!

Example

<table>
<thead>
<tr>
<th>Variable</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>34</td>
</tr>
</tbody>
</table>

Ignoring alignment issues
### Example

<table>
<thead>
<tr>
<th>Variable</th>
<th>Size</th>
<th>Offset</th>
<th>Offset with Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>26</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>34</td>
<td>48</td>
</tr>
</tbody>
</table>

4 bytes of padding
3 bytes of padding
7 bytes of padding

---

### Re-order!

*Place variables with most restrictive alignment first.*
**PCAT Example**

Initial offset: -4  
Increment: -4

<table>
<thead>
<tr>
<th>Variable</th>
<th>Size</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>4</td>
<td>-4</td>
</tr>
<tr>
<td>x</td>
<td>4</td>
<td>-8</td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>-12</td>
</tr>
<tr>
<td>z</td>
<td>4</td>
<td>-16</td>
</tr>
<tr>
<td>a</td>
<td>4</td>
<td>-20</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td>-24</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>-28</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>-32</td>
</tr>
</tbody>
</table>

%sp  
%fp-4  
%fp-8  
%fp-12  
%fp-16  
%fp-20  
%fp

**The Stack of Activation Records**

*An Abstract View...*

STACK TOP

Temp data fields  
Local data fields  
Machine Status  
Static link ("access link")  
Dynamic link ("control link")  
Return address  
Returned value (optional)  
Arguments ("actuals")

To access a local or temp variable...  
*(stacktop + offset)*

© Harry H. Porter, 2006
**The Dynamic Link**

When we return, we need to be able to go back to previous frame.

**During a call:**
- Save old StackTop
- Increment StackTop
  (Allocate new frame)
- Store old StackTop
  in Dynamic Link field

**During a return:**
- Use the Dynamic Link to restore the old StackTop

**To access local variables:**
\[ \text{StackTop} + \text{offset}_x \]

**To access variables in our caller’s frame:**
\[ ([\text{StackTop} + \text{offset}_{\text{DynamicLink}}] + \text{offset}_x) \]

**What do we need from the caller’s frame?**
- Arguments?
- Place to store a returned result?

---

**SPARC**

*Much of the Activation Record is cached in registers!!!*

**Dynamic Link**
- Stored in registers (%sp, %fp)

**Return Address**
- Stored in registers (%i7, %o7)

**Arguments**
- Some in registers %i0 ... %i5
- Rest in caller’s frame

**Returned Value**
- 32-bits: in register (%i0, %o0)

**Machine Status**
- 64 bytes
  - Architecture dependent & often not needed

© Harry H. Porter, 2006
The “Calling Sequence”

• Compute argument values
• Allocate new frame
• Initialize it
  • Move arguments into the new frame (optional)
  • Save machine state (optional)
  • Save return address
• Transfer control to new routine

The “Return Sequence”

• Compute and move return value (optional)
• Pop stack / delete the top frame
• Resume execution in the caller’s code

Flexibility as to who...

• caller
• callee

...does what...

• calling sequence
• return sequence

---

Caller’s Code:

```
... ...
... ...
call foo ...
... ...
```

Callee’s Code:

```
foo: ...
... ...
... ...
ret
```

---

Calling Sequence: BBB BBB BBB ...

Return Sequence: CCC CCC CCC
Where do you draw the line between the caller’s frame and the callee’s frame?

Callee must be responsible for part of the job since only it knows the size of its local area.

When compiling the caller...
The compiler does not have the callee’s code.
Parameter Passing in SPARC

Arguments to "bar" at known offset in the caller's frame

Using a Stack for Expression Evaluation

Source Code:

\[ x := y + (2 \times z); \]

Target Code:

push y
push 2
push z
mult
add
pop x
### Using a Stack for Expression Evaluation

**Source Code:**

\[ x := y + (2 \times z); \]

**Target Code:**

- push \( y \)
- push 2
- push \( z \)
- mult
- add
- pop \( x \)

---

### Using a Stack for Argument Evaluation and Parameter Passing

**Calling Sequence:**

- Push args onto the stack
- Save FP
- FP = TOP
- TOP = TOP + FrameSize

**Return Sequence:**

- Move return value to where arg 1 was
- Restore TOP, FP
- Pop stack top into...
Using a Stack for Argument Evaluation and Parameter Passing

**Calling Sequence:**
- Push args onto the stack
- Save FP
- FP = TOP
- TOP = TOP + FrameSize

**Return Sequence:**
- Move return value to where arg 1 was
- Restore TOP, FP
- Pop stack top into...

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

**Calling Sequence:**
- Push args onto the stack
- Save FP
- FP = TOP
- TOP = TOP + FrameSize

**Return Sequence:**
- Move return value to where arg 1 was
- Restore TOP, FP
- Pop stack top into...

---

**Source Code:**

```
x := y + 2 * (bar(7,i+1));
```

**Target Code:**

```
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x
```

---

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

**Source Code:**
\[
x := y + 2 * (\text{bar}(7,i+1));
\]

**Target Code:**
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x

Foo's Frame

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:
\[ x := y + 2 \ast (\text{bar}(7, i+1)); \]

Target Code:
- push \( y \)
- push 2
- push 7
- push \( i \)
- push 1
- add
- call \text{bar}
- mult
- add
- pop \( x \)

Foo’s Frame

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

**Source Code:**
\[ x := y + 2 \times \text{bar}(7,i+1); \]

**Target Code:**
- push y
- push 2
- push 7
- push i
- push 1
- add
- call bar
- mult
- add
- pop x

### Foo’s Frame
- \( x: 30 \)
- \( y: 30 \)
- \( z: 40 \)
- \( i: 50 \)

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

**Source Code:**
\[ x := y + 2 \ast (\text{bar}(7,i+1)); \]

**Target Code:**
- push \( y \)
- push \( 2 \)
- push \( 7 \)
- push \( i \)
- push \( 1 \)
- add
- call bar
- mult
- add
- pop \( x \)

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:
\[ x := y + 2 \ast (\text{bar}(7,i+1)); \]

Target Code:
push y
push 2
push 7
push i
push 1
add
call bar
mult
add
pop x

Foo's Frame

© Harry H. Porter, 2006
Using a Stack for Argument Evaluation and Parameter Passing

Source Code:
\[ x := y + 2 \times (\text{bar}(7,i+1)); \]

Target Code:
- push y
- push 2
- push 7
- push i
- push 1
- add
- call bar
- mult
- add
- pop x

```
[...]
```

Foo's Frame

FP
x: 36
y: 30
i: 50
z: 40

Goal:
Allow a routine to have variable-length data (i.e., dynamically-sized arrays) as local data in frame

© Harry H. Porter, 2006
Variable-Length Local Variables

**Goal:**
Allow a routine to have variable-length data
(i.e., dynamically-sized arrays) as local data in frame

**Option 1:**
Allocate the variable on the heap
Work with pointers to the data
PCAT: Hide the pointers from the programmer

Programmer codes:
\[ a[i] \]

Compiler produces code like this:
\[ *(a + 4*i) \]

Auto free the data when the routine returns?

**Option 2:**
Create the variable on the stack, dynamically
Effectively: Enlarge the frame as necessary
Still need to work with pointers
Variable-Length Local Variables

We must have two pointers:
Stack Top
FP

Local Variables
at fixed offsets from FP

Dynamically sized variables
use hidden pointers

All references to “a” and “b”
will be indirect
through hidden pointers

Array A
Array B

© Harry H. Porter, 2006
Local / Non-Local Variables

procedure main() {
    int y;
    procedure foo1() {
        int x;
        procedure foo2() {
            ...x...
            call foo3();
            ...y...
        }
        procedure foo3() {
            int x;
            ...x...
            call foo1 / call foo2
        }
        call foo1 / call foo2
            ...x...
    }
    call foo1
        ...y...
}
Local / Non-Local Variables

procedure main() {
  int y;
  procedure foo1() {
    int x1;
    procedure foo2() {
      int x2;
      call foo3();
      ...y...
    }
    procedure foo3() {
      int x3;
      call foo1 / call foo2
      ...x...
    }
  }
  call foo1 / call foo2
  ...x...
  call foo1
  ...y...
}

Local (0 levels)
Non-Local (1 level)
Non-Local (2 levels)
**Static Scoping Rules**

“Lexical Scoping”

For non-local variables...
- Look in syntactically enclosing routine
- Look in next enclosing routine
...etc...

```
procedure foo2
var x: integer

procedure foo3
var ...

procedure foo4
var ...
          ...
```

The “norm” for most languages

```
procedure foo2
    var x: integer

procedure foo3
    var ...

procedure foo4
    var ...
          ...
```

---

**Dynamic Scoping Rules**

For non-local variables...
- Search the calling stack at runtime to locate the right variable.

Uncommon.

```
procedure bar ()
begin
    ...
    x...
end

procedure foo1 ()
var x: integer
begin
    ...call bar()...
end

procedure foo2 ()
var x: integer
begin
    ...call bar()...
end
```

```
    bar

    fool
        x:

    fool
        x:
            ...
            ...
            ...
```
Statement Blocks

Syntax:

```
begin ... end;
{ ... }
```

Blocks are entered and exited in nested order.

**Idea:**

Create a new frame for each block.
Push on stack.
Statement Blocks

Blocks are entered and exited in nested order.

Idea:
Create a new frame for each block.
Push on stack.

But:
No parameters
No recursion
All calls are inline
⇒ Overhead!

So:
Just put variables in frame of surrounding routine!

Syntax:
begin ... end;
{ ... }

Frame for “foo”
Consider a language with:
Functions as objects
Non-local variable accesses

- Bar is called
  Bar's frame is created
  "x" is created
- Bar sets "p" to point to function "foo"
- Bar returns
  Bar's frame is popped
  "x" is destroyed
- foo is invoked
  foo accesses variable "x"

```
procedure main()
  var p: function
  ...
  p = ...
  ...
  call p()
  ...
endProcedure
```

```
procedure main()
  var p: function
  procedure bar()
    var x: ...
    procedure foo()
      ...
      x ...
      endProcedure
    ...
    p = foo
    ...
  endProcedure
  ...
  call bar()
  ...
  call p()
  ...
endProcedure
```
Consider a language with:
  Functions as objects
  Non-local variable accesses

• Bar is called
  Bar’s frame is created
  “x” is created
• Bar sets “p” to point to function “foo”
• Bar returns
  Bar’s frame is popped
  “x” is destroyed
• foo is invoked
  foo accesses variable “x”

Solution:
  Do not free bar’s frame
  ... until it is no longer needed
  Put bar’s frame on the heap
  Automatic garbage collection

```plaintext
procedure main()
  var p: function
  procedure bar()
    var x: ...
    procedure foo()
      ... x ...
      endProcedure
      ... p = foo
      ...
      endProcedure
      ...
      call bar()
      ...
      call p()
      ...
      endProcedure
```

The “C” Solution

“C” allows non-locals to be used within a function

However...
  • Functions may not be nested
  • Variables are either
    • Local
    • Global (i.e., static)

```plaintext
static int x;
void foo() {
  ... x ...
}
void bar () {
  ...
  p = &foo;
}
void main () {
  ...
  bar();
  ...
  (*p) ();
  ...
}
```
The “Static” Link

“Nesting Depth: A routine’s lexical level...
Main body ⇒ 0
Nested routines ⇒ Add one
Each frame contains a “static” link
Points to the frame of
the most recently invoked activation
of the lexically surrounding routine.
Given a variable usage...
   How do we find the frame containing the right variable?

Assume that \( x \) is declared at lexical level \( M \).

Assume that \( x \) is used at lexical level \( N \).
   (We must have \( N \geq M \))

At runtime...
   Follow \( N-M \) static links to find the right frame.
   Use the offset of \( x \) within that frame.

---

"x" declared at level M
"x" used at level N
Follow N-M static links
Code:
Follow static link 1 time
Use offset (e.g., -4) within that frame
How do we set up the Static Links?

Whenever we call a routine...
  must initialize static link in new frame.

Assume “foo” calls “bar”

\[ \text{Level} = L_{\text{FOO}} \]

\[ \text{Level} = L_{\text{BAR}} \]

Want to initialize bar’s static link.

What frame should it point to???
**Initializing the Static Link**

*foo calls bar*

**Goal:**
- Find the frame of the routine that lexically encloses bar
- Set bar’s static link to point to it.

**Given:**
- foo’s frame is on the stack, directly below the newly allocated frame for bar.

**Approach:**
- Use the static link in foo’s frame.
- Follow \( L_{\text{FOO}} - L_{\text{BAR}} + 1 \) static links from foo’s frame.

*This will be the frame of the routine that lexically encloses bar!!!*
- Make bar’s static link point to it.

---

**Case 1:**  
\( L_{\text{BAR}} = L_{\text{FOO}} + 1 \)  
*foo contains bar directly*

- proc foo
- proc bar
- ...call bar

---

**Case 2:**  
\( L_{\text{BAR}} = L_{\text{FOO}} \)  
*foo and bar at same level*

- proc foo
- proc bar
- ...call bar

---

**Case 3:**  
\( L_{\text{BAR}} < L_{\text{FOO}} \)  
*foo is more deeply nested*

- proc bar
- proc foo
- ...call bar

---

**Case 4:**  
\( L_{\text{BAR}} > L_{\text{FOO}} + 1 \)  
*bar is nested deeply*

- proc foo
- proc bar
- ...call bar

*Not Allowed!*
**Case 1:** \( L_{\text{BAR}} = L_{\text{FOO}} + 1 \)

*Foo statically contains bar directly.*

\[
L_{\text{FOO}} - L_{\text{BAR}} + 1
5 - 6 + 1 = 0
\]

From foo’s frame...
follow 0 links!
Just make bar’s static link point to foo’s frame.

---

**Case 2:** \( L_{\text{BAR}} = L_{\text{FOO}} \)

*Foo and bar at same level.*

\[
L_{\text{FOO}} - L_{\text{BAR}} + 1
6 - 6 + 1 = 1
\]

From foo’s frame...
follow 1 link!
Case 3: \( L_{\text{BAR}} < L_{\text{FOO}} \)

\( \text{foo is more deeply nested} \)
(\( \text{within bar or one of bar's siblings} \))

\[
\begin{align*}
\text{proc g} & \quad 5 \\
\text{proc bar} & \quad 6 \\
\text{proc foo} & \quad 9 \\
\ldots \text{call bar} & \\
\end{align*}
\]

\( L_{\text{FOO}} - L_{\text{BAR}} + 1 \)

\[9 - 6 + 1 = 4\]

From foo’s frame
follow 4 links!

Display Registers

The Idea:
In static storage...
maintain an array of pointers
\( d[\ldots] \)

The \( i \)-th element will be a pointer to an activation record on the stack...
...whose lexical level is \( i \).

Assume we are currently executing in a routine (“\( f \)”) at lexical level 6...

\( d[6] \) points to the top frame
(i.e., the currently executing frame, for “\( f \)”)

\( d[5] \) points to the most recent activation of the routine
that lexically encloses “\( f \)”.
(a routine at level 5, call it “\( g \)”)

\( d[4] \) points to the most recent activation of the routine
that lexically encloses “\( g \)”
(a routine at level 4, call it “\( h \)”)

\( \vdots \)

\( d[0] \) points to the most recent activation of a routine
at level 0, call it “main”
To access a non-local variable...

Go to $d[3]$ to get a pointer to the correct frame.

$d[i]$ where $i$ is the lexical level at which the variable was declared.

How to Maintain the Display Registers?

During “call” and “return” sequences...

Each activation record will have a word in which to save an old value of a display register.

"display register save area"

When calling a routine at lexical level “$i$”...

Allocate a new frame on the stack
Save old value of $d[i]$ in that word in the new frame
$d[i] := \text{ptr to the new frame}$

When returning...

$d[i] = \text{the saved value}$

Note: The entire array of display registers will always be restored to its previous value after any sequence of calls and matching returns!
Example: “f” calls “c”

- Allocate new frame
- Save old d[3]
- Set d[3] to point to new frame

© Harry H. Porter, 2006
Example: “f” calls “c”

• Allocate new frame
• Save old d[3]
• Set d[3] to point to new frame
Case 1:  
\[ L_{BAR} = L_{FOO} + 1 \]  
Assume “foo” calls “bar”  
\[ L_{LOOK} = L_{FOO} \]  
\[ L_{LOOK} = L_{BAR} \]  
Whatever “bar” does, these registers will be unchanged after “bar” returns  
foo contains bar directly  
procedure foo  
\[ \begin{array}{c} \text{procedure bar}  \\ \text{...call bar} \end{array} \]  
Routines that lexically enclose “foo”  
Whatever “bar” does, these registers will be unchanged after “bar” returns  

Cases 2 and 3:  
\[ L_{BAR} \leq L_{FOO} \]  
Routines that lexically enclose “foo” and “bar”  
procedure b  
\[ \begin{array}{c} \text{procedure c}  \\ \text{procedure d}  \\ \text{procedure e}  \\ \text{procedure foo}  \\ \text{...call bar} \end{array} \]  
procedure bar  
Routines that lexically enclose “foo” and “bar”  

Cases 2 and 3:

L\text{BAR} \leq L\text{FOO}

Routines that lexically enclose "foo" and "bar" display registers

Producing Target Code

At the beginning of the program

We’ll need to compute the maximum lexical level!

procEntry bar,lexLevel=4
returnVoid