Memory Locations For Variables
A Binding Question

- Variables are bound (dynamically) to values
- Those values must be stored somewhere
- Therefore, variables must somehow be bound to memory locations
- How?
Functional Meets Imperative

• Imperative languages expose the concept of memory locations: $a := 0$
  • Store a zero in $a$’s memory location
• Functional languages hide it: val $a = 0$
  • Bind $a$ to the value zero
• But both need to connect variables to values represented in memory
• So both face the same binding question
Outline

• Activation records
• Static allocation of activation records
• Stacks of activation records
• Handling nested function definitions
• Functions as parameters
• Long-lived activation records
Function Activations

- The lifetime of one execution of a function, from call to corresponding return, is called an activation of the function.
- When each activation has its own binding of a variable to a memory location, it is an activation-specific variable.
- (Also called dynamic or automatic)
Activation-Specific Variables

• In most modern languages, activation-specific variables are the most common kind:

```plaintext
fun fact n =  
  if (n=0) then 1  
  else  
    n * fact (n-1);
```

```plaintext
int fact(int n) { 
  if (n==0) return 1; 
  else  
    return n * fact(n-1); }
```
Block Activations

- For block constructs that contain code, we can speak of an activation of the *block*
- The lifetime of one execution of the block
- A variable might be specific to an activation of a particular block within a function:

```plaintext
fun fact n = 
  if (n=0) then 1 
else
  let
    val b = fact (n-1)
  in
    n*b 
end;
```

```plaintext
int fact(int n) {
  if (n==0) return 1;
  else
    {
      int b = fact(n-1);
      return n*b;
    }
}
```
Other Lifetimes For Variables

- Most imperative languages have a way to declare a variable that is bound to a single memory location for the entire runtime
- Obvious binding solution: static allocation (classically, the loader allocates these)

```c
int count = 0; // global scope
int nextcount() {
    return ++count;
}
```
Scope And Lifetime Differ

- In most modern languages, variables with local *scope* have activation-specific *lifetimes*, at least by default.

- However, these two aspects can be separated, as in C:

```c
int nextcount() {
    static int count = 0; // local scope
    count = count + 1;
    return count;
}
```
Other Lifetimes For Variables

- Object-oriented languages use variables whose lifetimes are associated with object lifetimes
- Some languages have variables whose values are persistent: they last across multiple executions of the program
- Will focus on activation-specific variables
Activation Records

• Language implementations usually allocate all the activation-specific variables of a function together as an *activation record*.

• The activation record also contains other activation-specific data, such as
  • Return address: where to go in the program when this activation returns
  • Link to caller’s activation record: more about this in a moment
Block Activation Records

• When a block is entered, space must be found for the local variables of that block

• Various possibilities:
  • Preallocate (static) in the containing function’s activation record
  • Extend the function’s activation record when the block is entered (and revert when exited)
  • Allocate separate block activation records

• Our illustrations will show the static option
Outline

• Activation-specific variables
• Static allocation of activation records
• Stacks of activation records
• Handling nested function definitions
• Functions as parameters
• Long-lived activation records
Static Allocation

- The simplest approach: allocate one activation record for every function, statically
- Older dialects of Fortran and Cobol used this system
- Simple and fast
Fortran Example

FUNCTION AVG (ARR, N)
DIMENSION ARR(N)
SUM = 0.0
DO 100 I = 1, N
   SUM = SUM + ARR(I)
100 CONTINUE
AVG = SUM / FLOAT(N)
RETURN
END

How are parameters passed?

<table>
<thead>
<tr>
<th>N address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARR address</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>SUM</td>
</tr>
<tr>
<td>AVG</td>
</tr>
</tbody>
</table>
Value and Reference Parameter Passing Review

x = 2;
y = 3;
switch(x, y);
:
void switch(
    float &a,
    float &b)
{
    float t;
    t = a;
    a = b;
    b = t;
}
Reference passing potentially dangerous

Passing literals by reference must be prevented

\[ x = 2 + 2; \]
\[ \text{three}(x); \]
\[ x = 2 + 2; \]

: 

void three(int &n) {
    n=3;
}

\textbf{Question}: How can this problem be prevented?
Exercise 1

What is the output of the following C++ program?

```cpp
void SUB(int &K, float &X) {
    K = 1;
    X = 20;
}

void main(void) {
    float A[2];
    int I;
    I = 0;
    A[0] = 10;
    SUB(I, A[I]);
}
```
Static Allocation

1. Simple, only one activation record per procedure.
2. Memory allocation done at load time.
3. Does not allow recursion. Why?
4. Does not allow for nested scope. Why?
5. Faster than dynamic allocation.
6. We’ll look at an example patterned after FORTRAN language.
Definition Key to Activation Records

1. **M** – Memory, a linear array $M[0],...,M[n]$.
2. **PAR** – Address of caller’s parameters, $PAR[1],...,PAR[n]$.
3. **IP** – Caller’s return address.
4. **TMP** – Temporary storage of Caller’s registers.
5. **DL** – Dynamic Link to Caller’s Activation Record.
6. **AR(S)** – The Activation Record address of function $S$. 
### Static Activation

**Caller Code for S**

\[ F(p_1,\ldots,p_n) \Rightarrow \]

\[ M[AR(S)].TMP = \text{REGs} \]
\[ M[AR(S)].IP = \text{resume} \]
\[ M[AR(F)].PAR[1] = \&p_1 \]
\[ \ldots \]
\[ M[AR(F)].PAR[n] = \&p_n \]

\[ M[AR(F)].DL = AR(S) \]

\[ \text{goto entry}(F) \]

**resume:**

\[ \text{REGs} = M[AR(S)].TMP \]

**Callee Code for F return**

\[ \text{goto } M[M[AR(F)].DL].IP \]

<table>
<thead>
<tr>
<th>AR(S)</th>
<th>Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S resume Line 4</td>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>AR(caller)</td>
<td>DL</td>
<td></td>
</tr>
<tr>
<td>S Regs</td>
<td>TMP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AR(F)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&amp;p_1</td>
<td>PAR[1]= &amp;p_1</td>
</tr>
<tr>
<td>y</td>
<td>&amp;p_2</td>
<td>PAR[2]= &amp;p_2</td>
</tr>
<tr>
<td>z</td>
<td>&amp;p_3</td>
<td>PAR[3]= &amp;p_3</td>
</tr>
</tbody>
</table>

1. void S(void) {
2.      int p1, p2, p3;
3.      F(p1, p2, p3);
4. }
5. void F(int &x, int &y, int &z) {
6.      z = x + y;
7. }
Compiler Writes CALL and RETURN Code

a) \( M[\text{AR( MAIN )}].\text{IP} = 3 \) // Resume address
b) \( M[\text{AR( P )}].\text{Par}[1] = 246 \) // X reference
c) \( M[\text{AR( P )}].\text{DL} = 102 \) // AR(MAIN) Addr
d) \( M[\text{AR( MAIN )}].\text{Tmp} = \text{Registers} \)
e) Goto Algorithm address 4 // Entry of P
f) \( \text{Registers} = M[\text{AR(MAIN)}].\text{Tmp} \)

1. \( X = 5; \)
   VOID MAIN {
2.     CALL P(X)
3.     PRINT, X
   }
4. VOID P(Y){
5.   Y = 12
6.   RETURN
   }

a) goto \( M[ M[\text{AR( P )}].\text{DL }].\text{IP} \) // Return address
   where \( M[\text{AR( P )}].\text{DL} \) is 102
   then \( M[102].\text{IP} \) is 3, the return address

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>3</td>
</tr>
<tr>
<td>103</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>Main Registers</td>
</tr>
<tr>
<td>105</td>
<td>X Address 246</td>
</tr>
<tr>
<td>106</td>
<td>0</td>
</tr>
<tr>
<td>107</td>
<td>102</td>
</tr>
<tr>
<td>108</td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
</tr>
</tbody>
</table>

Chapter Twelve  Modern P1
Parameter Access in Subprogram

Parameters are passed by reference through the `PAR[ ]` parameter list.

For `Y = 12` the compiler generates `M[ M[AR( P )].PAR[1] ] = 12`

`AR( P )` is 105

for `M[ AR( P )].PAR[1]` then `M[ 105 ].PAR[1]` is 246

for `M[ M[ AR( P ).PAR[1]] ]` then `M[ 246 ]` is 5 which is altered to 12.
```c
void main(void)
{
    float x=1400;
    float y=1600;
    float a;
    Ave(x, y, a);
}

void Ave(
    float &m,
    float &n,
    float &u)
{
    float z;
    Sum(m, n, z);
    u = z / 2.0;
}

void Sum(
    float &r,
    float &s,
    float &t)
{
    float w;
    w = r + s;
    t = w;
}
```

<table>
<thead>
<tr>
<th>Addr</th>
<th>main</th>
<th>Memory</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>11</td>
<td>IP</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>276</td>
<td>DL</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>277</td>
<td>TMP</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>278</td>
<td></td>
<td>100</td>
<td>PAR[1]=&amp;x</td>
</tr>
<tr>
<td>279</td>
<td></td>
<td>101</td>
<td>PAR[2]=&amp;y</td>
</tr>
<tr>
<td>280</td>
<td></td>
<td>102</td>
<td>PAR[3]=&amp;a</td>
</tr>
<tr>
<td>104</td>
<td></td>
<td>275</td>
<td>IP</td>
</tr>
<tr>
<td>12</td>
<td>275</td>
<td>275</td>
<td>DL</td>
</tr>
<tr>
<td>13</td>
<td>277</td>
<td>277</td>
<td>TMP</td>
</tr>
<tr>
<td>14</td>
<td>278</td>
<td>278</td>
<td></td>
</tr>
<tr>
<td>284</td>
<td></td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>285</td>
<td>285</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>286</td>
<td></td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>287</td>
<td>NA</td>
<td>IP</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>288</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>289</td>
<td></td>
</tr>
</tbody>
</table>

### Caller Code for S

\( F(p_1, ..., p_n) \rightarrow \)

- \( M[AR(S)].TMP = \text{REG} \)
- \( M[AR(S)].IP = \text{resume} \)
- \( M[AR(F)].PAR[1]= &p_1 \)
  - \( M[AR(F)].PAR[n]= &p_n \)
  - \( M[AR(F)].DL=AR(S) \)
  - \( \text{goto entry(F)} \)

**resume:**

\( \text{REG}=M[AR(S)].\text{TMP} \)

### Callee Code for F

\( \text{goto M[M[AR(F)].DL].IP} \)
Drawbacks to Static AR Allocation

- Each function has one activation record
- There can be only one activation alive at a time
- Modern languages (including modern dialects of Cobol and Fortran) do not obey this restriction:
  - Recursion
  - Multithreading
  - Nested scopes
Outline

• Activation-specific variables
• Static allocation of activation records
• Stacks of activation records
• Handling nested function definitions
• Functions as parameters
• Long-lived activation records
Stacks Of Activation Records

- To support recursion, we need to allocate a new activation record for each activation
- Dynamic allocation: activation record allocated when function is called
- For many languages, like C, it can be deallocated when the function returns
- A stack of activation records: stack frames pushed on call, popped on return
Current Activation Record

- STATIC: location of activation record was determined before runtime
- DYNAMIC: location of the current activation record is not known until runtime
- A function must know how to find the address of its current activation record
- Often, a special machine register (eBp on Intel) holds current activation record address
C Example

We are evaluating \texttt{fact(3)}. This shows the contents of memory just before the recursive call that creates a second activation.
This shows the contents of memory just before the third activation.

```c
int fact(int n) {
    int result;
    if (n<2) result = 1;
    else result = n * fact(n-1);
    return result;
}
```

![Diagram showing activation records and return addresses.](image)
This shows the contents of memory just before the third activation returns.

```c
int fact(int n) {
    int result;
    if (n<2) result = 1;
    else result = n * fact(n-1);
    return result;
}
```

<table>
<thead>
<tr>
<th>n: 1</th>
<th>Previous activation record</th>
<th>Result: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return address</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n: 2</th>
<th>Previous activation record</th>
<th>Result: ?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return address</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n: 3</th>
<th>Previous activation record</th>
<th>Result: ?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return address</td>
<td></td>
</tr>
</tbody>
</table>
The second activation is about to return.

```c
int fact(int n) {
    int result;
    if (n<2) result = 1;
    else result = n * fact(n-1);
    return result;
}
```

```
<table>
<thead>
<tr>
<th>n: 1</th>
<th>return address</th>
<th>previous activation record</th>
<th>result: 1</th>
</tr>
</thead>
</table>
```
```
<table>
<thead>
<tr>
<th>n: 2</th>
<th>return address</th>
<th>previous activation record</th>
<th>result: 2</th>
</tr>
</thead>
</table>
```
```
<table>
<thead>
<tr>
<th>n: 3</th>
<th>return address</th>
<th>previous activation record</th>
<th>result: ?</th>
</tr>
</thead>
</table>
```
The first activation is about to return with the result \texttt{fact(3) = 6}.

\begin{verbatim}
int fact(int n) {
    int result;
    if (n<2) result = 1;
    else result = n * fact(n-1);
    return result;
}
\end{verbatim}

\begin{tabular}{|c|c|}
\hline
\textbf{fact(1)} & \textbf{fact(2)} \\
\hline
\textbf{n: 1} & \textbf{n: 2} \\
\hline
return address & return address \\
\hline
previous activation record & previous activation record \\
\hline
\textbf{result: 1} & \textbf{result: 2} \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\textbf{fact(3)} \\
\hline
\textbf{n: 3} \\
\hline
return address \\
\hline
previous activation record \\
\hline
\textbf{result: 6} \\
\hline
\end{tabular}
ML Example

We are evaluating `halve [1,2,3,4]`. This shows the contents of memory just before the recursive call that creates a second activation.

```ml
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) = 
     let
       val (x, y) = halve cs 
     in
       (a::x, b::y)
     end;
```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;

This shows the contents of memory just before the third activation.
fun halve nil = (nil, nil)
|    halve [a] = ([a], nil)
|    halve (a::b::cs) =
    let
        val (x, y) = halve cs
    in
        (a::x, b::y)
    end;

This shows the contents of memory just before the third activation returns.

<table>
<thead>
<tr>
<th>current activation record</th>
</tr>
</thead>
<tbody>
<tr>
<td>halve []</td>
</tr>
<tr>
<td>parameter: []</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>previous activation record</td>
</tr>
<tr>
<td>value to return: ([], [])</td>
</tr>
</tbody>
</table>

| halve [3,4]               |
| parameter: [3,4]          |
| return address            |
| previous activation record|
| a: 3                      |
| b: 4                      |
| cs: []                    |
| x: ?                      |
| y: ?                      |
| value to return: ?        |

| halve [1,2,3,4]           |
| parameter: [1,2,3,4]      |
| return address            |
| previous activation record|
| a: 1                      |
| b: 2                      |
| cs: [3,4]                 |
| x: ?                      |
| y: ?                      |
| value to return: ?        |
The second activation is about to return.

fun halve nil = (nil, nil)
| halve [a] = ([a], nil)
| halve (a::b::cs) = 
  let 
    val (x, y) = halve cs 
  in 
    (a::x, b::y) 
  end;
The first activation is about to return with the result

\[
\text{halve } [1,2,3,4] = ([1,3],[2,4])
\]

```ml
fun halve nil = (nil, nil)
| halve [a] = ([a], nil)
| halve (a::b::cs) = 
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

**Current Activation Record**

- Parameter: []
- Previous Activation Record: [3,4]
- Return Address: 
- Value to Return: ([], [])

**Previous Activation Record**

- Parameter: [1,2,3,4]
- Previous Activation Record: [3,4]
- Return Address: 
- Value to Return: ([1,3],[2,4])

**Previous Activation Record**

- Parameter: [3,4]
- Previous Activation Record: []
- Return Address: 
- Value to Return: ([3], [4])

**Previous Activation Record**

- Parameter: [a]
- Previous Activation Record: [3,4]
- Return Address: 
- Value to Return: ([a], [])
Exercise 3

Diagram to deepest call:

fun power x 0 = 1
|   power x e = x * power x (e-1);
power 3 2;

int power (int x, int e) {
    if (e == 0) return 1;
    else return x * power( x, e-1);
}

power (3, 2);
Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
Nesting Functions

- What we just saw is adequate for many languages, including C
- But not for languages that allow:
  - Function definitions can be nested inside other function definitions
  - Inner functions can refer to local variables of the outer functions (under the usual block scoping rule)
- Like ML, Ada, Pascal, etc.
C++ Scope Nested

```cpp
a: int x = 5;
{  b: int y = x + 3;
    { c: int z = x + y + 4;
    }
  }
}
```

current activation record

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x: 5</td>
<td>dynamic link</td>
</tr>
<tr>
<td>b</td>
<td>y: 8</td>
<td>dynamic link</td>
</tr>
<tr>
<td>c</td>
<td>z: 17</td>
<td>dynamic link</td>
</tr>
</tbody>
</table>
```
Example – Nested Functions

Other variables accessible through nested scope?

fun quicksort nil = nil
| quicksort (pivot::rest) =
  let
    fun split(nil) = (nil,nil)
    | split(x::xs) =
      let
        val (below, above) = split(xs)
      in
        if x<pivot then (x::below, above)
        else (below, x::above)
      end;
    val (below, above) = split(rest)
  in
    quicksort below @ [pivot] @ quicksort above
  end;
The Problem

- How can an activation of the inner function (split) find the activation record of the outer function (quicksort)?
- It isn’t necessarily the previous activation record, since the caller of the inner function may be another inner function
- Or it may call itself recursively, as split does…
**Dynamic link** points to **caller’s** activation record

- **current activation record**
- **a split activation**
  - parameter
  - return address
  - previous activation record
  - **split’s variables:** \( x, xs, \text{ etc.} \)

- **another split activation**
  - parameter
  - return address
  - previous activation record
  - **split’s variables:** \( x, xs, \text{ etc.} \)

- **first caller: a quicksort activation**
  - parameter
  - return address
  - previous activation record
  - **quicksort’s variables:** pivot, rest, etc.
Nesting Link

• An inner function needs to be able to find the address of the most recent activation for the outer function

• We can keep this nesting link in the activation record…
Chapter Twelve

Modern Programming Languages

47
Setting The Nesting Link

- Easy if there is only one level of nesting:
  - Calling outer function: set to null
  - Calling from outer to inner: set nesting link same as caller’s activation record
  - Calling from inner to inner: set nesting link same as caller’s nesting link

- More complicated if there are multiple levels of nesting…
Multiple Levels Of Nesting

- References at the same level (\texttt{f1 to v1, f2 to v2, f3 to v3}) use current activation record
- References \( n \) nesting levels away chain back through \( n \) nesting links
- \textbf{Static Link} - Points to activation record of \textit{enclosing block}.
- \textbf{Dynamic Link} - Points to activation record of \textit{caller}.
Static Nesting Definitions

- **Activation record** – Contains local variables, parameters, links, etc.
- **ep** – *Environment pointer* to current activation record.
- **ip** – *Instruction pointer* to the current instruction.
- **Dynamic link** – Points to the calling function’s activation record.
- **Static link** – Points to the enclosing environment’s activation record. Represents the non-local data accessible to the function.
- **Static chain** – The static links from one enclosing environment to another.
- **SNL (Static Nesting Level)** – The number of enclosing environments where a symbol is defined or used.
- **SD (Static Distance)** – Difference between the SNL of *definition* and SNL of *use*, more intuitively, the number of static links in the static chain. SD to local data is 0, SD to nearest enclosing function is 1, etc.
- **Symbol table** – In statically nested environments, table recording symbol name, data type, SNL, and offset within the activation record. Used at compile time to generate code to access data bound to symbol.
Static Nesting

1. Void a() {
2.     int N;
3.     N = 1;
4.     b(19.3);
5. }

6. Void b(float sum) {
7.     int i;
8.     float avg;
9.     float Data[2];
10.    N = 2;
11.    c(5.8);
12. }

13. Void c(float val) {
14.     cout << val;
15. }

16. Void main() {
17.     a();
18. }

Chapter Twelve  Modern Programming Languages  51
```cpp
void main()
{
    void a()
    {
        int N;
        void b()
        {
            float sum;
            int i;
            float avg;
            float Data[2];
            void c()
            {
                cout << sum;
                cout << N;
                a();
            }
            N = 2;
            c(5.8);
        }
        N = 1;
        b(19.3);
    }
    a();
}
```
Static Nesting Calculations

1. void main()
2. {
3.   void a()
4.   {
5.     int N;
6.     void b(
7.       float sum)
8.     {
9.       void c(
10.          float val)
11.       {
12.         cout << sum;
13.         cout << N;
14.         a();
15.       }
16.       N = 2;
17.       c(5.8);
18.     }
19.   }
20.   N = 1;
21.   b(19.3);
22. }
23. a();

SNL Definition
2 Line 3 N
3 Line 5 sum
1 Line 2 a
4 Line 7 val

SNL Use
2 Line 15 N
4 Line 9 N
4 Line 8 sum
4 Line 10 a

SD = SNL Use - SNL Definition
0 Line 15 SD N = 2 - 2
1 Line 8 SD sum = 4 - 3
2 Line 9 SD N = 4 - 2
3 Line 10 SD a = 4 - 1

Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>SNL</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>int</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sum</td>
<td>float</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>val</td>
<td>float</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
1. void main()
2. {
3.     int N;
4.     void a()
5.     {
6.         void b(
7.             float val)
8.         {
9.             cout << val;
10.            cout << N;
11.            a();
12.         }
13.     }
14.     N = -9;
15.     b(5.8);
16. }
17.     N = 6;
18.     a();
19. }

Level 0

Level 1

Level 2

Level 3
Chapter Twelve

Modern Programming Languages

1. void main()
   { int N;
   void a()
   { void b(float val)
      { cout << val;
      cout << N;
      a();
      }
   }
   N = -9;
   b(5.8);
   N = 6;
   a();
   }

Exercise: Compute the access of:
N=-9;
at Line 10.

General access to variable:
1. ap=ep;
2. for (i=1; i<=SD(v); i++) ap=M[ap].SL;
3. access M[ap+offset(v)];

Example: \texttt{cout}<<\texttt{N} at Line 7, SD(N)=2:
1. ap = ep = 422
2. ap = M[ap].SL=M[422].SL=419
   ap = M[ap].SL=M[419].SL=415
3. cout<<M[ap+offset(N)]=M[415+0]=-9

Stack | Address
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>419</td>
<td>SL 425</td>
</tr>
<tr>
<td>9</td>
<td>IP 424</td>
</tr>
<tr>
<td>419</td>
<td>DL 423</td>
</tr>
<tr>
<td>5.8</td>
<td>val 422</td>
</tr>
<tr>
<td>415</td>
<td>SL 421</td>
</tr>
<tr>
<td>12</td>
<td>IP 420</td>
</tr>
<tr>
<td>415</td>
<td>DL 419</td>
</tr>
<tr>
<td>15</td>
<td>IP 417</td>
</tr>
<tr>
<td>OS</td>
<td>DL 416</td>
</tr>
<tr>
<td>-9</td>
<td>N 415</td>
</tr>
</tbody>
</table>

Line 7 SD(N)=2
Line 10 SD(N)=1
Line 13 SD(N)=0
Exercise 4

1. void a()
2. {
3.     int N;
4.     void b()
5.     {
6.         N = -2;
7.         c(5.8);
8.     }
9.     void c(
10.         float val)
11.     {
12.         cout << val;
13.         cout << N;
14.     }
15.     N = -1;
16.     b();
17. }

General access to variable:
1. ap=ep;
2. for (i=1; i<=SD(v); i++) ap=M[ap].SL;
3. access M[ap+offset(v)];

1. Static nesting level of definition:
   a) N    __
   b) b    __
   c) c     __
   d) val  __

2. Static distance of identifiers at lines:
   a) Line 12, N __
   b) Line 10, N __
   c) Line 9, val __

3. Fill in stack information through line 10 execution.

4. Compute the access to N at Line 10.

Stack Address
|   | 428 |
|   | 427 |
|   | 426 |
|   | 425 |
|   | 424 |
|   | 423 |
|   | 422 |
|   | 421 |
|   | 428 |
|   | 427 |
|   | 426 |
|   | 425 |
|   | 424 |
|   | 423 |
|   | 422 |
|   | 421 |
|   | 420 |
|   | 419 |
|   | 418 SL
| AR (a) | 417 IP
|   | 416 DL
|   | 415 N
Other Solutions

• The problem: references from inner functions to variables in outer ones
  • Nesting links in activation records: as shown
  • Displays: nesting links not in the activation records, but collected in a single static array
  • Lambda lifting: problem references replaced by references to new, hidden parameters
Outline

- Activation-specific variables
- Static allocation of activation records
- Stacks of activation records
- Handling nested function definitions
- Functions as parameters
- Long-lived activation records
Functions As Parameters

• When you pass a function as a parameter, what really gets passed?

• Code must be part of it: source code, compiled code, pointer to code, or implementation in some other form

• For some languages, something more is required…
Exercise 5 - C++ Example

1. void p(int x) {
2.   cout << "p " << 2*x;
3. }

4. void t(int x) {
5.   cout << "t " << x*x;
6. }

7. void q( void fp(int), int x) {
8.   fp(x);
9. }

10. void main(void) {
11.   q( p, -4 );
12.   q( t, -5 );
13. }

1. Trace lines executed.
2. What is the output?
C++ Example – Function Parameters

1. void p(int x) {
2.     cout << "p " << 2*x;
3. }

4. void t(int x) {
5.     cout << "t " << x*x;
6. }

7. void q( void fp(int), int x) {
8.     fp(x);
9. }

10. void main( void) {
11. q( p, -4 );
12. q( t, -5 );
13. }

Without nested environments, only the function address is passed as a parameter.

Execution of Line 11.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>426</td>
</tr>
<tr>
<td>AR(p)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;p = 1</td>
<td>422   PAR[1]</td>
</tr>
<tr>
<td>AR(q)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>417</td>
</tr>
<tr>
<td>AR(main)</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>OS</td>
</tr>
</tbody>
</table>
Exercise 6 - Function Parameters

1. void p(int x) {
2.    cout << "p " << 2*x;
3. }
4. void t(int x) {
5.    cout << "t " << x*x;
6. }
7. void q( void fp(int), int x ) {
8.    fp(x);
9. }
10. void main(void) {
11.    q( p, -4 );
12.    q( t, -5 );
13.}

Fill in the stack for execution starting at line 12.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>426</td>
</tr>
<tr>
<td></td>
<td>425 PAR[1]</td>
</tr>
<tr>
<td></td>
<td>424 IP</td>
</tr>
<tr>
<td></td>
<td>423 DL</td>
</tr>
<tr>
<td>AR(q)</td>
<td>422 PAR[1]</td>
</tr>
<tr>
<td></td>
<td>421 PAR[2]</td>
</tr>
<tr>
<td></td>
<td>420 IP</td>
</tr>
<tr>
<td></td>
<td>419 DL</td>
</tr>
<tr>
<td>AR(main)</td>
<td>418 IP</td>
</tr>
<tr>
<td></td>
<td>OS</td>
</tr>
</tbody>
</table>
ML Example

fun addXToAll (x, theList) = 
  let
    fun addX y = y + x;
  in
    map addX theList
  end;

fun map f [] = 
  | map f (h::t) =
    (f h)::(map f t);

- This function adds \( x \) to each element of \( \text{theList} \)
- Notice: \texttt{addXToAll} calls \texttt{map}, \texttt{map} calls \texttt{addX},
  and \texttt{addX} refers to a variable \( x \) in \texttt{addXToAll}'s activation record
Nesting Links Again

- When `map` calls `addX`, what nesting link will `addX` be given?
  - Not `map`’s activation record: `addX` is not nested inside `map`
  - Not `map`’s nesting link: `map` is not nested inside anything

- To make this work, the parameter `addX` passed to `map` must include the nesting link to use when `addX` is called
Not Just For Parameters

- Many languages allow functions to be passed as parameters
- Functional languages allow many more kinds of operations on function-values:
  - passed as parameters, returned from functions, constructed by expressions, etc.
- Function-values include both code to call, and nesting link to use when calling it
fun addXToAll \((x, \text{theList})\) =
  let
    fun addX \(y\) = \(y + x\);
  in
    map addX theList
  end;

This shows the contents of memory just before the call to \texttt{map}. The variable \texttt{addX} is bound to a function-value including code and nesting link.
C++ Style Example

1. void q ( void fp ( ) ) {
2.   fp ( );
3. }

4. void main ( void ) {
5.   int x = -5;
6.   void p ( ) {
7.     cout << x;
8. }
9.   q ( p );
10. }

In a nested environment, passing functions as parameters requires passing:
1. a reference to the enclosing environment.
2. Address of function parameter.

In the example, Line 7 accesses x, with SD(x) = 1.
Outline

• Activation-specific variables
• Static allocation of activation records
• Stacks of activation records
• Handling nested function definitions
• Functions as parameters
• Long-lived activation records
One More Complication

• What happens if a function value is used after the function that created it has returned?

```ml
fun funToAddX x = 
  let 
    fun addX y = 
      y + x;
  in 
  addX 
end;

fun test = 
  let 
    val f = funToAddX 3; 
  in 
    f 5 
  end;
```

```ml
fun funToAddX x = 
  let 
    fun addX y = 
      y + x;
  in 
    addX 
end;
```
fun test = 
  let 
    val f = funToAddX 3; 
  in 
    f 5 
  end;

fun funToAddX x = 
  let 
    fun addX y = 
      y + x; 
  in 
    addX 
  end;

This shows the contents of memory just before funToAddX returns.
fun test =  
  let  
    val f = funToAddX 3;  
  in  
    f 5  
  end;  

fun funToAddX x =  
  let  
    fun addX y =  
      y + x;  
  in  
    addX  
  end;  

fun funToAddX x =  
  let  
    fun addX y =  
      y + x;  
  in  
    addX  
  end;  

test;  

After funToAddX  

returns, f is the bound to  

test calls f which is \( y \rightarrow y + x \)  

To access \( x=3 \) in test must link to activation  

record for funToAddX that is already finished  

Fails if the language system deallocated that  

activation record when funToAddX returned
The Problem

- When `test` calls `f`, the function will use its nesting link to access `x`
- That is a link to an activation record for an activation that is finished
- This will fail if the language system deallocated that activation record when the function returned
C++ Example – Locals in AR

```cpp
int * ThreeInts() {
    int ti[5] = {10, 11, 12};
    for (int i = 0; i < 3; i++) cout << ti[i];
    return ti;
}

void main() {
    int * TI = ThreeInts();
    for (int i = 0; i < 3; i++) cout << TI[i];
}
```

What is the output?
public class Example {
    public static void main(String a[]) {
        int TI[] = ThreeInts();
        for (int i=0; i<3; i++)
            System.out.print(TI[i]);
    }

    public static int[] ThreeInts() {
        int ti[] = {10,11,12};
        for (int i=0; i<3; i++)
            System.out.print(ti[i]);
        return ti;
    }
}

What is the output?
The Solution

- For ML, and other languages that have this problem, activation records cannot always be allocated and deallocated in stack order.
- Even when a function returns, there may be links to its activation record that will be used; it can’t be deallocated if it is reachable.
- *Garbage collection*: chapter 14, coming soon!
Conclusion

• The more sophisticated the language, the harder it is to bind activation-specific variables to memory locations
  • Static allocation: works for languages that permit only one activation at a time (like early dialects of Fortran and Cobol)
  • Simple stack allocation: works for languages that do not allow nested functions (like C)
Conclusion, Continued

- Nesting links (or some such trick): required for languages that allow nested functions (like ML, Ada and Pascal); function values must include both code and nesting link

- Some languages (like ML) permit references to activation records for activations that are finished; so activation records cannot be deallocated on return