Chapter 10

Implementing Subprograms
Chapter 10 Topics

• The General Semantics of Calls and Returns
• Implementing “Simple” Subprograms
• Implementing Subprograms with Stack-Dynamic Local Variables
• Nested Subprograms with static scoping
• Blocks
• Implementing Dynamic Scoping
The General Semantics of Calls and Returns

• The subprogram call and return operations of a language are together called its subprogram linkage

• A subprogram call has numerous actions associated with it
  – Parameter passing methods
  – Static local variables
  – Execution status of calling program
  – Transfer of control
  – Subprogram nesting
Implementing “Simple” Subprograms: Calls and Returns

• **Call** Semantics
  – Save the execution *status* of the caller
  – Carry out the *parameter*-passing process
  – Pass the *return address* to the callee
  – Transfer control to the callee

• **Return** Semantics
  – If pass-by-value-result parameters are used, move the current values of those *parameters* to their corresponding actual parameters
  – If it is a function, move the *functional value* to a place the caller can get it
  – Restore the execution status of the caller
  – Transfer control back to the caller
Implementing “Simple” Subprograms

- Two separate parts: the code and the noncode part (local variables and data that can change)
- The layout of the noncode part of an executing subprogram is called an activation record
  - An activation record for simple subprogram

| Local variables | Parameters | Return address |

- An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation)
Code and Activation Records of a Program with “Simple” Subprograms
Implementing Subprograms with Stack-Dynamic Local Variables

• More complex activation record
  – The compiler must generate code to cause implicit allocation and de-allocation of local variables
  – Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)
Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is static, but its size may be dynamic.
- The *dynamic link* points to the top of an instance of the activation record of the caller.
- An activation record instance is dynamically created when a subprogram is called.
- Run-time stack

<table>
<thead>
<tr>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Dynamic link</td>
</tr>
<tr>
<td>Return address</td>
</tr>
</tbody>
</table>

Stack top
An Example: C Function

```c
void sub( float total, int part )
{
    int list[5];
    float sum;
    ...
}
```

```
| Local     | sum  |
| Local     | list [4] |
| Local     | list [3] |
| Local     | list [2] |
| Local     | list [1] |
| Local     | list [0] |
| Parameter | part   |
| Parameter | total  |
| Dynamic link | Return address |
```
An Example Without Recursion

```c
void A(int x) {
    int y;
    ...
    C(y);
}
void B(float r) {
    int s, t;
    ...
    A(s);
}
void C(int q) {
    ...
}
void main() {
    float p;
    ...
    B(p);
}
```

Call sequence:
- main calls B
- B calls A
- A calls C
An Example Without Recursion

ARI = activation record instance

ARI for fun1
Local
Parameter
Dynamic link
Return (to fun2)
Local

ARI for fun2
Local
Parameter
Dynamic link
Return (to main)
Local

ARI for main
Local

ARI for fun3
Parameter
Dynamic link
Return (to fun1)
Local

Copyright © 2006 Addison-Wesley. All rights reserved.
Dynamic Chain and Local Offset

• The collection of dynamic links in the stack at a given time is called the *dynamic chain*, or *call chain*.

• Local variables can be accessed by their offset from the beginning of the activation record. This offset is called the *local_offset*.

• The local_offset of a local variable can be determined by the compiler at compile time.
An Example With Recursion

The activation record used in the previous example supports recursion, e.g.

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

void main() {
    int value;
    value = factorial(3);
}
```

![Activation record for factorial](image)
int factorial (int n) {
    if(n <= 1) return 1; else return (n * factorial(n - 1));
}
void main() {
    int value; value = factorial(3);
}

ARI: Activation Record Instance
Stack content at position 1 in `factorial`

```c
int factorial (int n) {
    if(n <= 1) return 1; else return (n * factorial(n - 1));
}
void main() {
    int value; value = factorial(3);
}
```

Third call

---

Copyright © 2006 Addison Wesley. All rights reserved.
Stack content at position 2 in factorial

First ARI for factorial

- Functional value
- Parameter 1
- Dynamic link
- Return (to factorial)

Second ARI for factorial

- Functional value
- Parameter 2
- Dynamic link
- Return (to factorial)

Third ARI for factorial

- Parameter 1
- Dynamic link
- Return (to factorial)

ARI for main

- Functional value
- Parameter 3
- Dynamic link
- Return (to main)

At position 2 in factorial

second call completed

first call completed

third call completed
Stack content at position 3 in factorial

```c
int factorial (int n) {
    if(n <= 1) return 1; else return (n * factorial(n - 1));
}
void main() {
    int value;
    value = factorial(3);
}
```

In position 3 in main
final results
Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, JavaScript) use stack-dynamic local variables and allow subprograms to be nested.
- All variables that can be non-locally accessed reside in some activation record instance in the stack.
- The process of locating a non-local reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance
Locating a Non-local Reference

• Finding the offset is easy

• Finding the correct activation record instance
  – Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made
Static Scoping

- **static chain**: a chain of static links that connect certain activation record instances (ARI).
- During the execution of a subprogram A, the static link of its ARI points to an ARI of A's static parent. That instance's static link points in turn to A's static grandparent, if there is one.
- So, the static chain links all the static ancestors of an executing subprogram, in order of static parent first.
Static Scoping

• **static depth**: an integer associated with a static scope that indicates how deeply it is nested in the outermost scope.
  – An Ada main procedure has a static depth of 0.
  – A procedure A defined in main has static depth of 1.
  – A nested procedure B defined in A has static depth of 2.

• **chain offset**: the number of links to the correct ARI for a nonlocal reference
  – The difference between static_depth of the procedure containing the reference to X and the static_depth of the procedure containing the declaration for X.
Static Scoping

• Chain_offset example:
  – Static_depth of A, B, and C are 0, 1, and 2, respectively
  – If procedure C references a variable declared in A, the chain_offset is 2
  – If procedure C references a variable declared in B, the chain_offset is 1.
  – References to locals has chain_offset of 0.

Procedure A is
Procedure B is
Procedure C is

... end; --- of C
... end; --- of B
... end; --- of A
program MAIN_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin {SUB1}
        A := B + C;  \<------ 1
      end; {SUB1}
    procedure SUB2(X: integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin {SUB3}
          SUB1;
          E := B + A;  \<------ 2
        end; {SUB3}
        begin {SUB2}
          SUB3;
          A := D + E;  \<------ 3
        end; {SUB2}
        begin {BIGSUB}
          SUB2(7);
        end; {BIGSUB}
  begin
    BIGSUB;
  end; { MAIN_2 }
Stack Contents

• A Non-local reference can be represented by a pair of (chain_offset, local_offset)
  – chain_offset: number of links to the correct ARI.
  – local_offset: offsets from the beginning of the ARI.

• The references to variable A at points 1, 2, and 3, can be represented by
  – (0, 3): local
  – (2, 3): two levels away
  – (1, 3): one level away
Blocks

- Blocks are user-specified local scopes for variables
- An example in C
  
  ```c
  { 
    int temp;
    temp = list [upper];
    list [upper] = list [lower];
    list [lower] = temp
  }
  ```
  
- The lifetime of `temp` in the above example begins when control enters the block
- An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name
Implementing Blocks

• Two Methods:

1. Treat blocks as parameter-less subprograms that are always called from the same location
   – Every block has an activation record; an instance is created every time the block is executed

2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record
**Blocks**

```c
void main() {
    int x, y, z;
    while(...) {
        int a, b, c;
        ...
        while(...) {
            int d, e;
            ...
        }
    }
    while(...) {
        int f, g;
        ...
    }
    ...
}
```
Implementing Dynamic Scoping

• *Deep Access*: non-local references are found by searching the activation record instances on the dynamic chain

• *Shallow Access*: put locals in a central place
  – One stack for each variable name
  – Central table with an entry for each variable name
Void sub3() {
    int x, z;
    x = u + v;
    ...
}
Void sub2() {
    int w, x;
    ...
}
Void sub1() {
    int v, w;
    ...
}
Void main() {
    int v, u;
    ...
}

• The ARIs do not have static links which have no use for dynamic-scoped language

The ARIs do not have static links which have no use for dynamic-scoped language
Deep Access

• Reference to $x$ in sub3
  – found in the ARI of sub3
• Reference to $u$ in sub3
  – found in the ARI of main
  – searching four dynamic links and examining ten variable names
• Reference to $v$
  – found in the ARI of sub1
  – searching two dynamic links and examining five variable names
Deep Access

• Two differences between the deep access method for dynamic-scoped language and the static chain method for static-scoped languages
  
  — For dynamic scoped language, no way to determine at compile time the length of the chain that must be searched. Every activation record instance in the chain must be searched until the variable is found.
  
  — For dynamic scoped language, activation records must store the names of variables for the search process, whereas in static-scoped language implementations only the values are required (all variables are represented by chain_office/local_offset pairs which are determined at compile time).
Shallow Access

Void sub3() {
    int x, z;
    x = u + v;
    ...
}

Void sub2() {
    int w, x;
    ...
}

Void sub1() {
    int v, w;
    ...
}

Void main() {
    int v, u;
    ...
}

- Adv: fast references to variables
- Disadv: maintaining stacks at the entrances and exits of subprograms is costly
Summary

• Subprogram linkage semantics requires many action by the implementation
• Subprograms with stack-dynamic local variables and nested subprograms have two components
  — actual code and activation record
• Activation record instances contain formal parameters and local variables among other things
• Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
• Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or central variable table method