Binding and Scoping

Read: Scott, Chapter 3.1, 3.2 and 3.3.1, 3.3.2 and 3.3.6

Lecture Outline

- *Notion of binding time*
- *Object lifetime and storage management*

- *Scoping*
  - *Static scoping*
  - *Dynamic scoping*

Notion of Binding Time

- Binding time (Scott): the time an answer becomes associated to an open question

3

Notion of Binding Time

- Static
  - Before program execution
- Dynamic
  - During program executes

4

Examples of Binding Time Decisions

- Binding time (Scott): the time an answer becomes associated to an open question
- Binding a variable name to a memory location
  - Static or dynamic
  - Determined by scoping rules
- Binding a variable/expression to a type
  - Static or dynamic
- Binding a call to a target subroutine
  - Static (as it is in C, mostly) or dynamic (virtual calls in Java, C++)

5

5

Example: Binding Variables to Locations

- Map a variable to a location
  - Map variable at use to a location
  - Map subroutine at use to target subroutine
- Determined by scoping rules
  - Static scoping
    - Binding before execution
  - Dynamic scoping
    - Binding during execution
- More on scoping later…

```
int x,y;
void foo(int x)
{
   y = x;
   int y = 0;
   if (y) {
     int y;
     y = 1;

   }
}
```


6

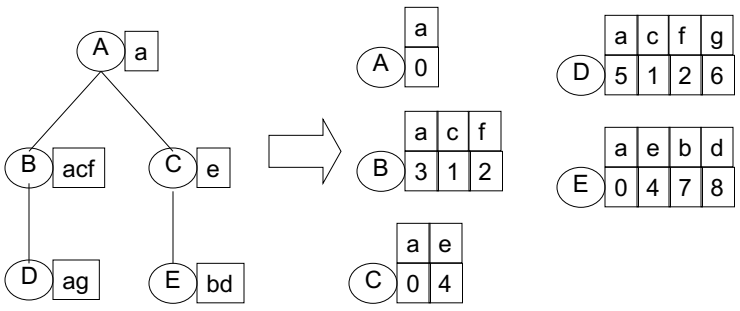
6

General View of Dynamic Binding

- Dynamic binding
  - What are the advantages of dynamic binding?
  - Disadvantages?
- An example: Cost of dynamic binding of call to target method in OO languages

Example: Cost of Dynamic Dispatch in C++

- Source: Driesen and Hölzle, OOPSLA'96



Virtual function tables (VFTs)
Capital characters denote classes,
lowercase characters message selectors,
and numbers method addresses

```
load [object_reg+#VFTOffset],table_reg
load [table_reg+#selectorOffset],method_reg
call method_reg
```

Extra instructions: cost extra

Other Choices Related to Binding Time

- Pointers: introduce "heap variables"
  - Good for flexibility – allows dynamic structures
  - Bad for efficiency – direct cost: accessed indirectly; indirect cost: compiler unable to perform optimizations
- Most PLs support pointers
  - Issues of management of heap memory
    - Explicit allocation and deallocation
    - Implicit deallocation (garbage collection)
- PL design choices – many subtle variations
  - No pointers (FORTRAN 77)
  - Explicit pointers (C++ and C)
  - Implicit pointers (Java)

Lecture Outline

- *Notion of binding time*
- *Object lifetime and storage management*
- *Scoping*
  - *Static scoping*
  - *Dynamic scoping*

Storage Allocation Mechanisms

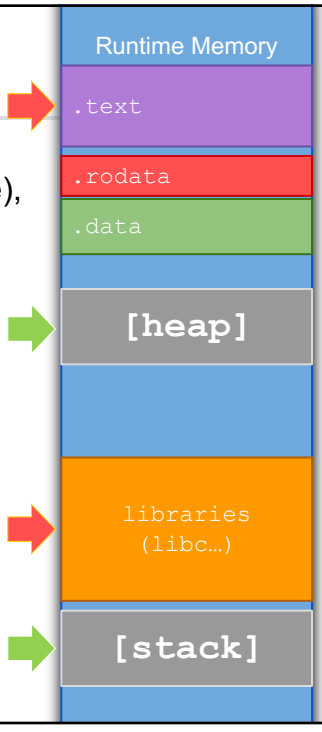
- Static storage – an object is given absolute address, which is the same throughout execution
  - What is an example of static data?
- Stack storage – stack objects are allocated on a run-time stack at subroutine call and deallocated at return
  - Needs a stack management algorithm
  - What is an example of stack data?
- Heap storage - long-lived objects are allocated and deallocated at arbitrary times during execution
  - Needs the most complex storage management algorithm

Combined View

- Static storage: .text (program code), .rodata, .data, etc.

- Stack contains one stack frame per executing subroutine
- Stack grows from higher towards lower memory addresses

- Heap contains objects allocated and not yet de-allocated
- Heap grows from lower towards higher memory addresses



Memory graph courtesy of RPISEC/MBE class.

Examples of Static Data

- Program code
- Global variables
- Tables of type data (e.g., inheritance structure)
- Dispatch tables (VFTs) and other tables
- Other

Examples of Stack Data

- What data is stored on the stack?
- Local variables, including parameters
- Compiler-generated temporaries (i.e., for expression evaluation)
- Bookkeeping (stack management) information
- Return address

Run-time Stack

- Stack contains frames of all subroutines that have been entered and not yet exited from
- Frame contains all information necessary to update stack when subroutine is exited
- Stack management uses two pointers: fp (**f**rame **p**ointer) and sp (**s**tack **p**ointer)
  - fp points to a location at the start of current frame
    - In higher memory (but lower on picture)
  - sp points to the next available location on stack (or the last used location on some machines)
    - In lower memory (but higher up on picture)
  - fp and sp define the beginning and the end of the frame

Run-time Stack

Run-time Stack Management

- Addresses for local variables are encoded as sp + offset
  - But may also have fp - offset
- Idea:
  - When subroutine is entered, its frame is placed on the stack. sp and fp are updated accordingly
  - All local variable accesses refer to this frame
  - When subroutine is exited, its frame is removed from the stack and sp and fp are updated accordingly

Frame Details

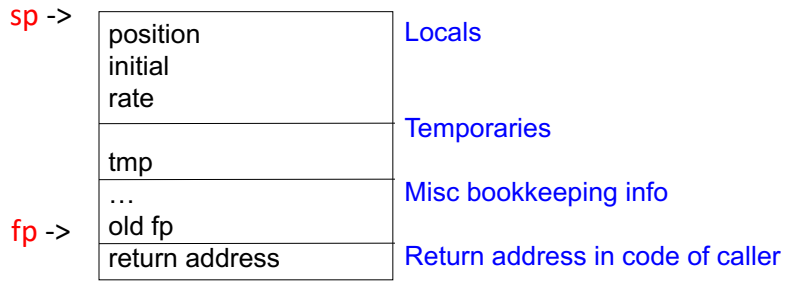
- Arguments to called routines
- Local variables
- Temporaries
- Miscellaneous bookkeeping information
  - Saved address of start of caller's frame (old fp)
  - Saved state (register values of caller), other
- Return address

Frame Example

```
void foo(double rate, double initial) {
  double position; …
  position = initial + rate*60.0; …
  return;
}
```

Assume **bar** calls **foo**.

Frame for **foo:**



19

Lecture Outline

- *Notion of binding time: The time an answer becomes associated with an open question*
  - *Static, before program execution*
  - *Dynamic, during program execution*
- *Object lifetime and storage management*
- *Scoping*
  - *Static scoping*
  - *Dynamic scoping*

20

Scoping

- Scoping rules answer the binding time question: how do we map a variable to a location (declaration)?
- Most languages use static scoping
  - Mapping from variable to location (declaration) is decided before program execution
- Block-structured programming languages
  - Nested subroutines (Pascal, Scheme, Python, etc.)
  - Nested blocks (C, C++ { … })
- Scope: region of program text where a declaration is visible

Static Scoping in Block Structured Programming Languages

- Also known as lexical scoping
- Block structure gives rise to scopes and the closest nested scope rule
  - There are local variable declarations within a block
  - A block inherits variable declarations from enclosing blocks
  - Local declarations take precedence over inherited ones
    - Hole in scope of inherited declaration
    - In other words, inherited declaration is hidden
- Lookup for non-local variables proceeds from inner to outer enclosing blocks

Example Block-Structured PL

```
main
    a, b, c: integer
    procedure P
            c: integer
        procedure S
                c, d: integer
            procedure R
            …
            end R
            R()
        end S
        R()
        S()
    end P
    procedure R
            a: integer
            … = a, b, c
    end R
     P()
end main
```

main.a, main.b, P.c — main.P, P.S, main.R

main.a,main.b,S.c,S.d — main.P, P.S, S.R

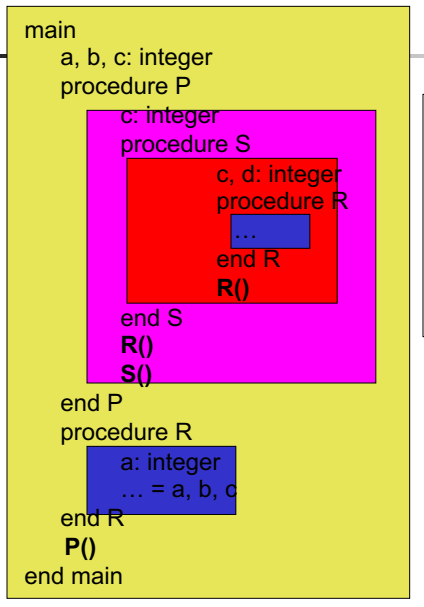
main.P, P.S, S.R

R.a, main.b, main.c — main.R, main.P

Nested block structure allows locally defined variables and subroutines

23

23

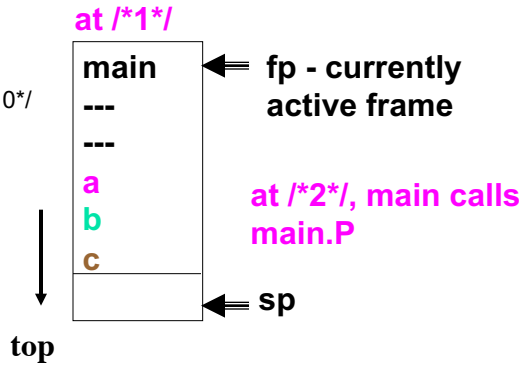


Rule: a variable is visible if it is declared in its own block or in a textually surrounding block **and** it is not 'hidden' by a declaration in a closer block (i.e., there is no hole in scope)

24

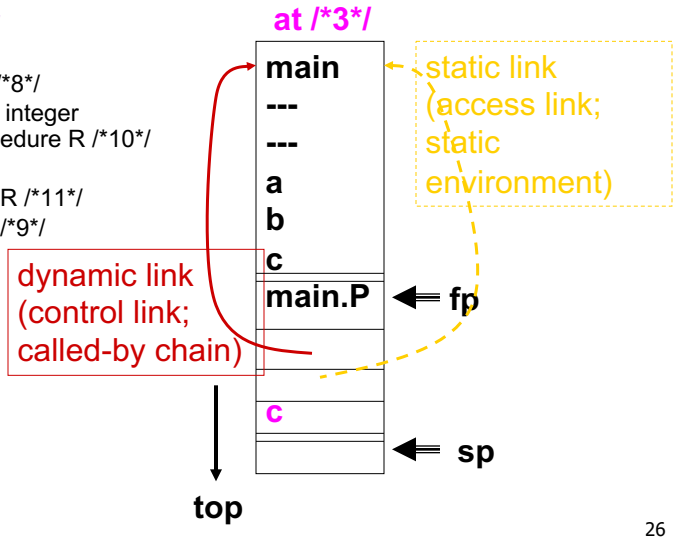
Example with Frames

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

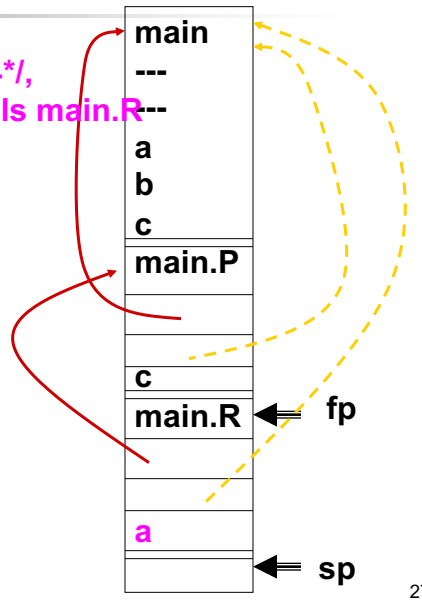


Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

at /*5*/

at /*4*/,
P calls main.R



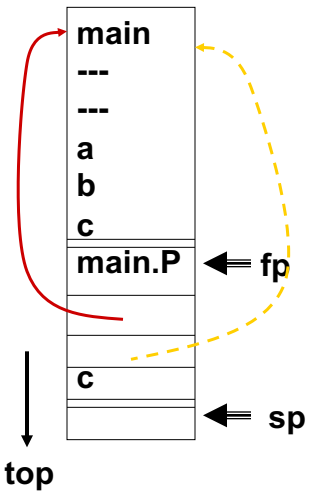
27

27

Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

at /*6*/ main.R exits
sp ← fp
fp ← old fp
in main.R's frame



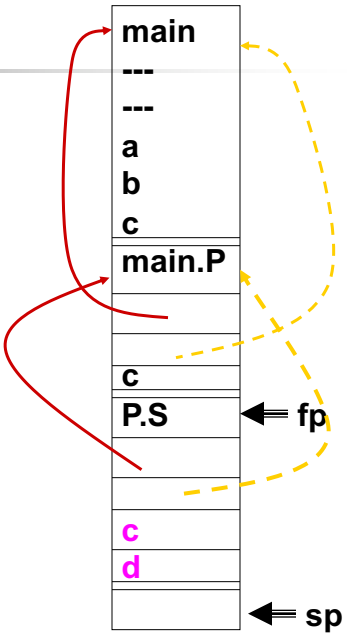
28

28

Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

at /*7*/, P calls P.S; at /*8*/:



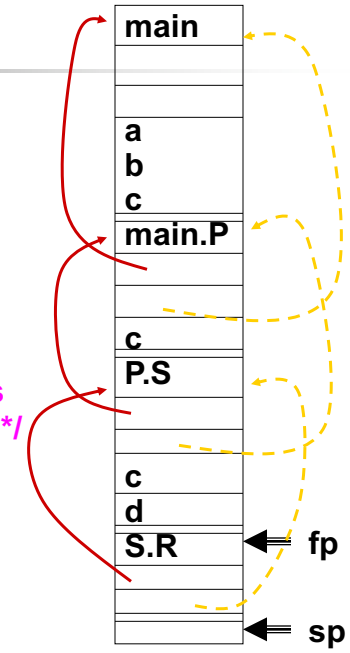
29

29

Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

at /*9*/ S calls in S.R; at /*10*/



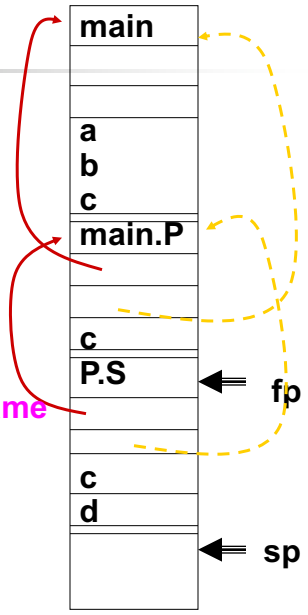
30

30

Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

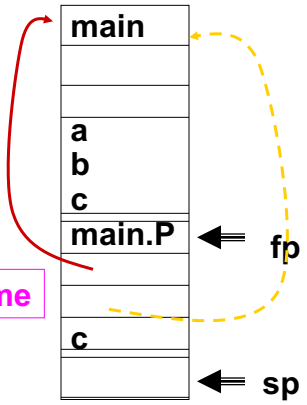
/*11*/ pop S.R's frame



Example

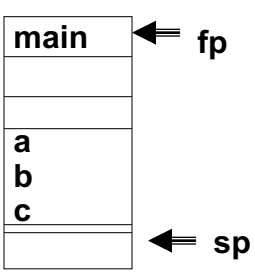
```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

/*12*/pop S's frame



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
            c: integer
            procedure S /*8*/
                    c, d: integer
                    procedure R /*10*/
                    …
                    end R /*11*/
                    R()  /*9*/
            end S /*12*/
            R() /*4*/
            S() /*7*/
    end P /*13*/
    procedure R /*5*/
            a: integer
            … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```



/*13*/ pop P's frame
/*14*/ pop main's frame
so that sp ← fp

Static Link vs. Dynamic Link

- Static link for a frame of subroutine P points to the most recent frame of P's lexically enclosing subroutine
  - Bookkeeping required to maintain the static link
  - If P is *k*-blocks deep from main, then there is a **chain of static links** of length *k* from P to main
  - To find non-local variables, follow chain of static links
- Dynamic link points to the caller frame, this is just the old fp pointer stored on the frame

Observations

- Static link of P points to the frame of the most recent frame of Q, where Q is the lexically enclosing subroutine of P
- Dynamic link of P may point to a different subroutine's frame, depending on where P is called from

An Important Note!

- For now, we assume languages where subroutines are third-class values, meaning that subroutines cannot be passed as arguments, returned or assigned to variables
- When subroutines are third-class values, it is guaranteed that when P is called, a frame of its lexically enclosing subroutine is on the stack (i.e., P's static reference environment is available)
  - A subroutine cannot outlive its reference environment

An Important Note!

- In languages that allow subroutines to be passed as arguments and returned from other subroutines, i.e., subroutines (functions) are first-class values, scoping rules become more involved
- We will return to scoping when we discuss functional languages

Dynamic Scoping

- Allows for local variable declaration
- Inherits non-local variables from subroutines that are live when current subroutine is invoked
  - Use of variable is resolved to the declaration of that variable in the most recently invoked and not yet terminated frame. I.e., lookup proceeds from closest predecessor on stack to furthest
  - (old) Lisp, APL, Snobol, Perl

Example

```
main
    procedure Z
        a: integer
        a := 1
        Y()
        output a
    end Z
    procedure W
        a: integer
        a := 2
        Y()
        output a
    end W
    procedure Y
        a := 0 /*1*/
    end Y
    Z()
    W()
end main
```

Which a is modified at /*1*/ under dynamic scoping? Z.a or W.a or both?

39

39

Example

```
main
    procedure Z
        a: integer
        a := 1
        Y()
        output a
    end Z
    procedure W
        a: integer
        a := 2
        Y()
        output a
    end W
    procedure Y
        a := 0; /*1*/
    end Y
    Z()
    W()
end main
```

main calls Z,
Z calls Y,
Y sets Z.a to 0.

40

40

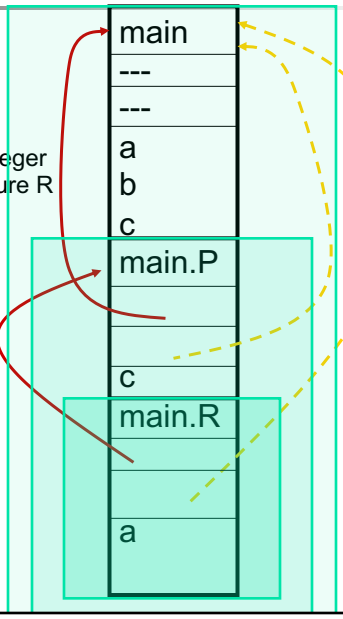
Example

```
main
    procedure Z
        a: integer
        a := 1
        Y()
        output a
    end Z
    procedure W
        a: integer
        a := 2
        Y()
        output a
    end W
    procedure Y
        a := 0; /*1*/
    end Y
    Z()
    W()
end main
```

main calls W,
W calls Y,
Y sets W.a to 0.

Static vs. Dynamic Scoping

```
main
    a, b, c: integer
    procedure P
        c: integer
        procedure S
            c, d: integer
            procedure R
            …
            end R
            R()
        end S
        R()
        S()
    end P
    procedure R
        a: integer
        … = a, b, c
    end R
    P()
end main
```



Static Scoping:
a bound to R.a,
b to main.b,
c to main.c

Dynamic Scoping:
a bound to R.a,
b to main.b,
c to P.c

Dynamic Scoping

- Dynamic scoping is considered a bad idea. Why?
- More on static and dynamic scoping to come!

What is Python's Scoping Discipline?

```
def main():
  a = 0; b = 0; c = 0;
  def P():
    c = 1
    def S():
      c = 2; d = 2;
      def R():
        print("Calling red R")
      R()
    R()
    S()
  def R():
    a = 3
    print(a)
  P()
  def R():
    print("Calling blue R")
  P()
main()
```

The End