How to Create a Programming Language

LECTURE 7: INTRODUCTION TO SYMBOL MANAGEMENT

1

LECTURE OUTLINE

- BINDING
- LIFETIME
- SCOPE

2

BINDING

- AN ASSOCIATION BETWEEN A NAME AND AN OBJECT (DATA)
- DIFFERENT FROM OBJECT CREATION/DESTRUCTION
- BINDING TIME – WHEN THE BINDING OCCURS
  - EARLIEST: LANGUAGE DESIGN TIME
  - LATEST: RUN TIME
- STATIC VS DYNAMIC BINDING
- DEEP VS SHALLOW BINDING

3

BINDING TIME

- LANGUAGE DESIGN TIME
  - WHEN LANGUAGE IS FIRST DESIGNED
  - BASIC TYPES, CONTROL FLOW CONSTRUCTS, ETC.
- LANGUAGE IMPLEMENTATION TIME
  - IMPLEMENTATION SPECIFIC BEHAVIOR ALLOWED BY LANGUAGE STANDARD
  - EX: SIZE OF *LONG*, *SHORT*, *CHAR*, AND *INT* TYPE IN C
- COMPILE TIME
  - MAPPING OF SOME VARIABLES TO MEMORY
  - EX: C *STATIC* VARIABLES

4

BINDING TIME

- LINK TIME
  - LINKER RESOLVES INTER-MODULE REFERENCES
  - EX: C AUTOMATIC VARIABLES (OUTSIDE ANY FUNCTION) FROM SOURCE MODULE OUTSIDE CURRENT C FILE
- LOAD TIME
  - FINAL RUNTIME ADDRESS SELECTED BY 'LOADER' (MOSTLY IN EARLIER OS)
  - LESS COMMON TODAY

5

BINDING TIME

- RUN TIME
  - DURING EXECUTION – VERY BROAD
    - PROGRAM STARTUP
    - MODULE ENTRY
    - ELABORATION
    - SUBROUTINE CALL
    - BLOCK ENTRY
    - STATEMENT EXECUTION
  - EX: C AUTOMATIC VARIABLES INSIDE FUNCTION ALLOCATED ON A STACK FRAME

6

STATIC VS DYNAMIC BINDING

- STATIC BINDING
  - TYPICALLY, BINDINGS DECIDED PRIOR TO RUN TIME
  - BINDINGS IN COMPILED LANGUAGES TEND TO BE STATIC
  - MORE EFFICIENT AT RUN-TIME
- DYNAMIC BINDING
  - TYPICALLY, BINDINGS DECIDED AT RUN TIME
  - BINDINGS IN INTERPRETED LANGUAGES TEND TO BE DYNAMIC
  - LESS EFFICIENT, MORE FLEXIBLE

7

DEEP VS SHALLOW BINDING

- WITH DYNAMIC BINDING
  - REFERENCING ENVIRONMENT – VISIBLE VARIABLES WHEN A FUNCTION IS CALLED
  - SOME LANGUAGES ALLOW FUNCTION/PROCEDURE AS ARGUMENT
  - WHICH VARIABLE SET USED WHEN THAT FUNCTION/PROCEDURE ARGUMENT IS CALLED?

8

DEEP VS SHALLOW BINDING

```
int key_var = 5;

procedure b() {
  if (key_var > 2) {
    print "Cool" }

procedure a(callme:procedure) {
  int key_var = -2;
  callme(); }

a(b);        // Call a and ask it to call b
```


9

DEEP VS SHALLOW BINDING

```
int key_var = 5;

procedure b() {
  if (key_var > 2) {
    print "Cool!" }

procedure a(callme:procedure) {
  int key_var = -2;
  callme(); }

a(b);        // Call a and ask it to call b
```

Q: When 'b' is called, will it or will it not print Cool!

10

DEEP VS SHALLOW BINDING

```
int key_var = 5;

procedure b() {
  if (key_var > 2) {
    print "Cool!" }

procedure a(callme:procedure) {
  int key_var = -2;
  callme(); }

a(b);        // Call a and ask it to call b
```

Q: When 'b' is called, will it or will it not print Cool!

A: It Depends!

11

DEEP VS SHALLOW BINDING

```
int key_var = 5;

procedure b() {
  if (key_var > 2) {
    print "Cool!" }

procedure a(callme:procedure) {
  int key_var = -2;
  callme(); }

a(b);        // Call a and ask it to call b
```

Q: When 'b' is called, will it or will it not print Cool!

A: It Depends!

Deep binding occurs at the call to a() so the 'referencing' environment is a snapshot at this point in time.

12

DEEP VS SHALLOW BINDING

```
int key_var = 5;

procedure b() {
   if (key_var > 2) {
      print "Cool!" }

procedure a(callme:procedure) {
   int key_var = -2;
   callme(); }

a(b);        // Call a and ask it to call b
```

Q: When 'b' is called, will it or will it not print Cool!

A: It Depends!

Shallow binding occurs at the actual use of the symbol inside of a() so the 'referencing' environment is from the immediately surrounding procedure.

13

RELATED CONCEPTS

- DECLARATION VERSUS DEFINITION
- DECLARATION ORDER AND RECURSIVE TYPES

14

DECLARATION VERSUS DEFINITION

- **DECLARATION** – TELLS THE COMPILER THE NAME OF A VARIABLE
- **DEFINITION** – THIS DESCRIBES VARIABLE SUFFICIENTLY TO CREATE THE OBJECT AND ALLOCATE MEMORY

15

EXAMPLE: DECLARATION VERSUS DEFINITION

Which of the following are Declarations? Which are Definitions?

```
extern int anarray[10];
struct employee;

int a, b;
float c;
struct employee *eptr;
…


struct employee {
  char name[40];
  int type;
}
```


16

EXAMPLE: DECLARATION VERSUS DEFINITION

Declaration – insufficient to create objects

```
extern int anarray[10];
struct employee;

int a, b;
float c;
struct employee *eptr;
…


struct employee {
  char name[40];
  int type;
}
```


17

EXAMPLE: DECLARATION VERSUS DEFINITION

Declaration – insufficient to create objects

```
extern int anarray[10];
struct employee;

int a, b;
float c;
struct employee *eptr;
…


struct employee {
  char name[40];
  int type;
}
```

Definitions – Can allocate the objects since *int* and *float* types have known sizes for a particiular CPU. Also, all pointers on a specific CPU are the same size (i.e. 4 or 8 bytes)

Definition – With the field list included, the compiler can determine that this will need 40 bytes plus the size of an int type.

18

FOR THOUGHT: DECLARATION VERSUS DEFINITION

Why not just move the definition ahead of its use??

```
struct employee {
  char name[40];
  int type;
}

int a, b;
float c;
struct employee *eptr;
…
```


19

DECLARATION ORDER

- THE ORDER IN WHICH VARIABLES ARE DECLARED MAY BE IMPORTANT. QUESTIONS:
  - DOES A VARIABLE BINDING EXIST FOR THE WHOLE BLOCK IN WHICH IT IS DECLARED (WHOLE-BLOCK DECLARATION) OR ONLY FROM IT'S DECLARATION POINT FORWARD?
  - IF A VARIABLE IN AN OUTER BLOCK WITH THE SAME NAME EXISTS, WHICH VALUE IS USED FOR ASSIGNMENT AHEAD OF LOCAL DEFINITION?

20

DECLARATION ORDER EXAMPLE

```
const int a = 10;
int main(int argc, char **argv) {
  int b = a;
  int a = 5;
  printf ("b is %d, a is %d\n", b, a);
...
```

C – b gets the value of the 'global' symbol a

```
Program tb2;
Var
  a :Integer := 10;
Procedure tryit;
Var
  b : Integer := a;
  a : Integer := 5;

Begin
  Writeln("b=", b, ", a=", a);
End;
```

Pascal – This throws an error since Pascal uses the concept of 'whole-block' declaration

21

LIFETIME AND STORAGE ALLOCATION

- STORAGE ALLOCATION TYPES
  - STATIC
    - ABSOLUTE ADDRESS
    - DOES NOT CHANGE
  - STACK
    - ADDRESS ASSIGNED ENTERING A SCOPE LIKE A FUNCTION OR BLOCK
    - LAST-IN, FIRST-OUT
    - VARIABLE MAY NOT EXIST AT CERTAIN POINTS
  - HEAP
    - STORAGE ALLOCATED AD-HOC (USUALLY BY PROGRAMMER)
    - COMPLEX STORAGE MANAGEMENT SCHEME NEEDED

22

LIFETIME

- UNDERLYING CONCEPTS
  - CREATION OF OBJECTS
  - CREATION OF BINDINGS
  - USE OF REFERENCES
  - DEACTIVATION/REACTIVATION OF BINDINGS
  - DESTRUCTION OF BINDINGS
  - DESTRUCTION OF OBJECTS

Lifetime

- **LIFETIME** – THE TIME BETWEEN THE CREATION OF A BINDING AND THE DESTRUCTION OF A BINDING

23

LIFETIME

- ABILITY TO REFERENCE A VARIABLE MAY DIFFER FROM LIFETIME
  - REFERENCES
    - PARAMETERS IN SUBROUTINES
    - OKAY AS LONG AS VARIABLE STORAGE STILL ALLOCATED
  - DANGLING REFERENCES
    - STORAGE DEALLOCATED WHILE BINDING STILL ACTIVE
    - EX: REFERENCE TO LOCAL VARIABLE RETURNED TO CALLER

24

SCOPE

- THE TEXTUAL RANGE WHERE A NAME-TO-OBJECT BINDING IS 'ACTIVE'
- STATIC VS. DYNAMIC SCOPING
- ELABORATION

25

STATIC SCOPE

- STATIC SCOPE USES A NAME-TO-OBJECT BINDING FROM THE INNERMOST LEXICAL SCOPE
- CAN BE DETERMINED AT COMPILE TIME

26

STATIC SCOPE - EXAMPLE

```
Program testscope;
Var
   a : Integer := 10;
   b : Integer := 20;
Procedure tryit1;
Var
   a : Integer := 30;
   Procedure tryit2;
   var
      c : Integer := a;
      d : Integer := b;
   begin
      Writeln("c=", c, ", d=", d);
      a := 501;
      b := 601;
   end;
begin
   Writeln("b=", b, ", a=", a);
   tryit2;
   Writeln("b=", b, ", a=", a);
end; { testscope }
begin
   Writeln("Before tryit1 : a=", a, ", b=",
b);
   tryit1;
   Writeln("After tryit1 : a=", a, ", b=", b);
end.
```


27

STATIC SCOPE - EXAMPLE

```
Program testscope;
Var
   a : Integer := 10;
   b : Integer := 20;
Procedure tryit1;
Var
   a : Integer := 30;
   Procedure tryit2;
   var
      c : Integer := a;
      d : Integer := b;
   begin
      Writeln("c=", c, ", d=", d);
      a := 501;
      b := 601;
   end;
begin
   Writeln("b=", b, ", a=", a);
   tryit2;
   Writeln("b=", b, ", a=", a);
end; { testscope }
begin
   Writeln("Before tryit1 : a=", a, ", b=",
b);
   tryit1;
   Writeln("After tryit1 : a=", a, ", b=", b);
end.
```


28

STATIC SCOPE - EXAMPLE

```
Program testscope;
Var
  a : Integer := 10;
  b : Integer := 20;
Procedure tryit1;
Var
  a : Integer := 30;
  Procedure tryit2;
  var
    c : Integer := a;
    d : Integer := b;
  begin
    Writeln("c=", c, ", d=", d);
    a := 501;
    b := 601;
  end;
begin
  Writeln("b=", b, ", a=", a);
  tryit2;
  Writeln("b=", b, ", a=", a);
end; { testscope }
begin
  Writeln("Before tryit1 : a=", a, ", b=", b);
  tryit1;
  Writeln("After tryit1 : a=", a, ", b=", b);
end.
```


29

STATIC SCOPE - EXAMPLE

```
Program testscope;
Var
  a : Integer := 10;
  b : Integer := 20;
Procedure tryit1;
Var
  a : Integer := 30;
  Procedure tryit2;
  var
    c : Integer := a;
    d : Integer := b;
  begin
    Writeln("c=", c, ", d=", d);
    a := 501;
    b := 601;
  end;
begin
  Writeln("b=", b, ", a=", a);
  tryit2;
  Writeln("b=", b, ", a=", a);
end; { testscope }
begin
  Writeln("Before tryit1 : a=", a, ", b=", b);
  tryit1;
  Writeln("After tryit1 : a=", a, ", b=", b);
end.
```


30

DYNAMIC SCOPE

- DYNAMIC SCOPING USES NAME-TO-OBJECT BINDING FROM THE INNER MOST SCOPE IN THE RUN-TIME **CALL SEQUENCE**
- DIFFICULT OR IMPOSSIBLE TO RESOLVE AT COMPILE TIME
- RUN-TIME TYPE-CHECKING MAY REVEAL SEMANTIC ERRORS DUE TO CALL SEQUENCE

31

DYNAMIC SCOPE - EXAMPLE

```
$a = 10;

sub printVal()
{
   print "In printVal: ";
   print "a is: ", $a, "\n";
}

sub sub1()
{
   local $a;
   $a = 15;
   printVal();
}

print "Calling printVal\n";
printVal();
print "Calling sub1\n";
sub1();
print "Calling printVal again from main code\n";
printVal();
```


32

DYNAMIC SCOPE - EXAMPLE

```
$a = 10;

sub printVal()
{
    print "In printVal: ";
    print "a is: ", $a, "\n";
}

sub sub1()
{
    local $a;
    $a = 15;
    printVal();
}

print "Calling printVal\n";
printVal();
print "Calling sub1\n";
sub1();
print "Calling printVal again from main code\n";
printVal();
```


33

DYNAMIC SCOPE - EXAMPLE

```
$a = 10;

sub printVal()
{
    print "In printVal: ";
    print "a is: ", $a, "\n";
}

sub sub1()
{
    local $a;
    $a = 15;
    printVal();
}

print "Calling printVal\n";
printVal();
print "Calling sub1\n";
sub1();
print "Calling printVal again from main code\n";
printVal();
```


34

ELABORATION

- OCCURS AT RUN-TIME
- CREATION OF A BINDING
- ALSO, ALLOCATION OF SPACE FOR STACK VARIABLE
- ALSO, POSSIBLY, INITIALIZATION

35

QUESTIONS

36