CSE 304/504 Compiler Design Introduction to Symbol Management

Lecture Outline

Binding

Lifetime

Scope

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

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

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

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

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

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?

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
```

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!

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 vs Shallow Binding

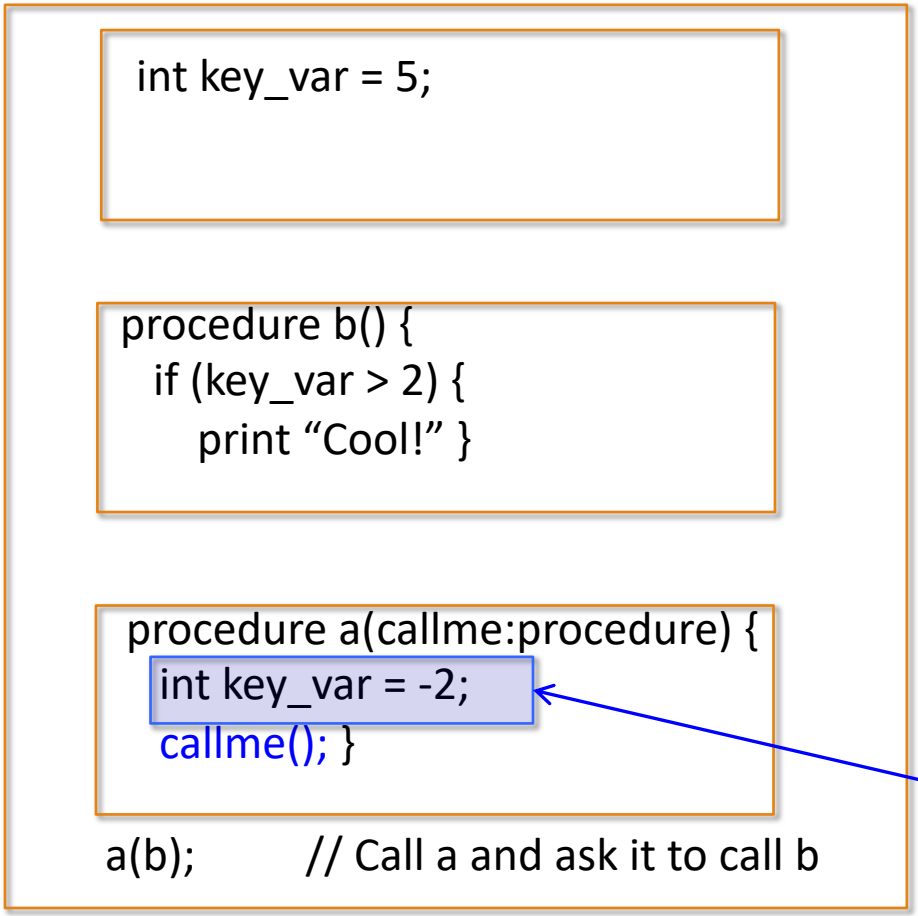
```
int key_var = 5;

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.

Deep vs Shallow Binding



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.

Related Concepts

Declaration versus Definition

Declaration Order and Recursive Types

Declaration versus Definition

Declaration – Tells the compiler the name of a variable

Definition – This describes variable sufficiently to create the object and allocate memory

Exercise: 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;
}
```

Exercise: Declaration versus Definition


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

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


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

Exercise: Declaration versus Definition

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

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


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

Declaration – insufficient to create objects

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.

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;
…
```

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?

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

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

Lifetime

Underlying concepts

- Creation of Objects
- Creation of bindings
- Use of references
- Deactivation/reactivation of bindings
- Destruction of bindings
- Destruction of objects

Lifetime – The time between the creation of a binding and the destruction of a binding

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

Scope

The textual range where a name-to-object binding is ‘active’

Static vs. Dynamic Scoping

Elaboration

Static Scope

Static scope uses a name-to-object binding from the innermost lexical scope

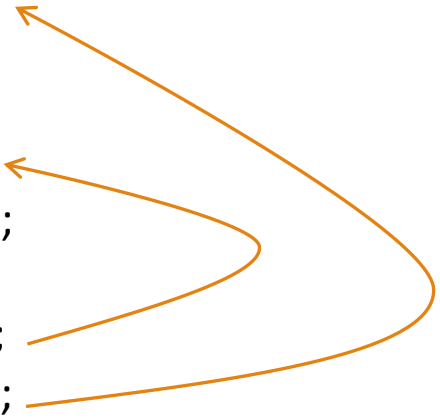
Can be determined at compile time

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.
```

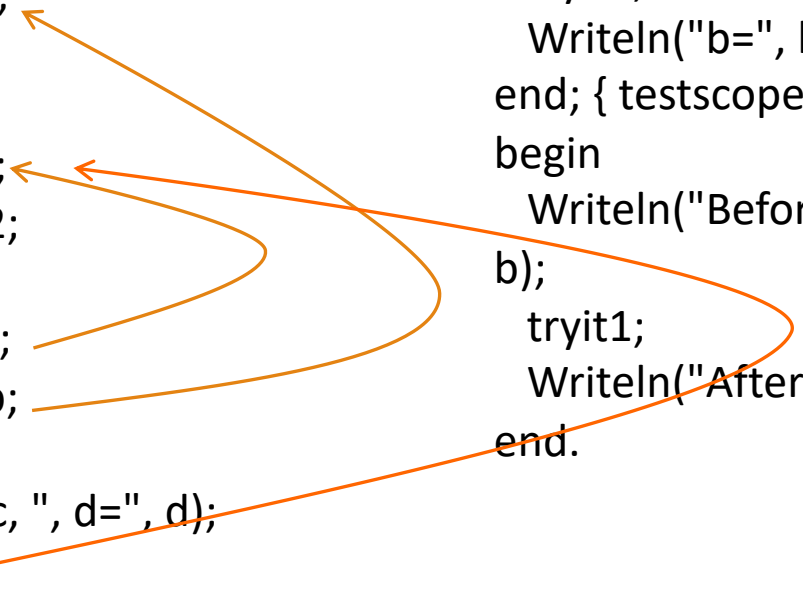
Static Scope - Example

```
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.
```



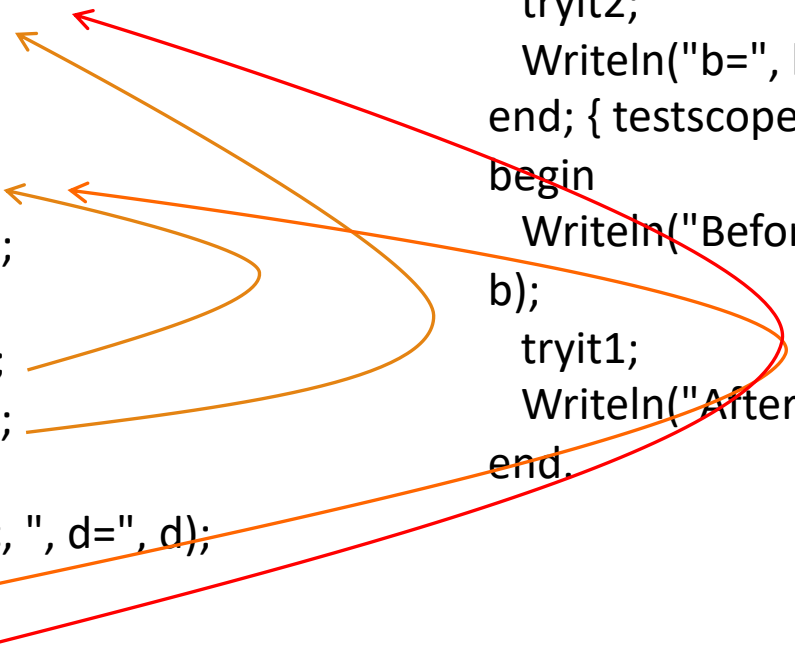
Static Scope - Example

```
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.
```



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.
```



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

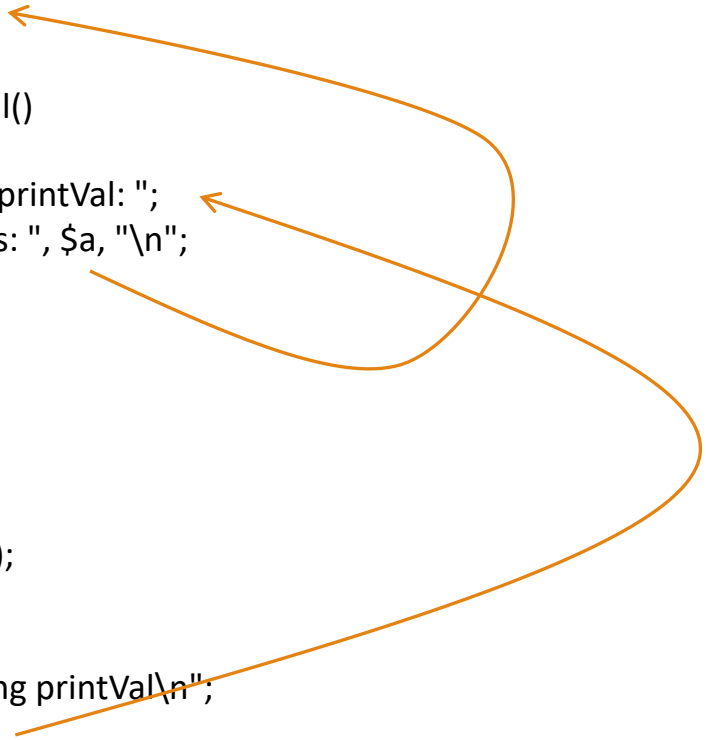
Run-time type-checking may reveal semantic errors due to call sequence

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();
```

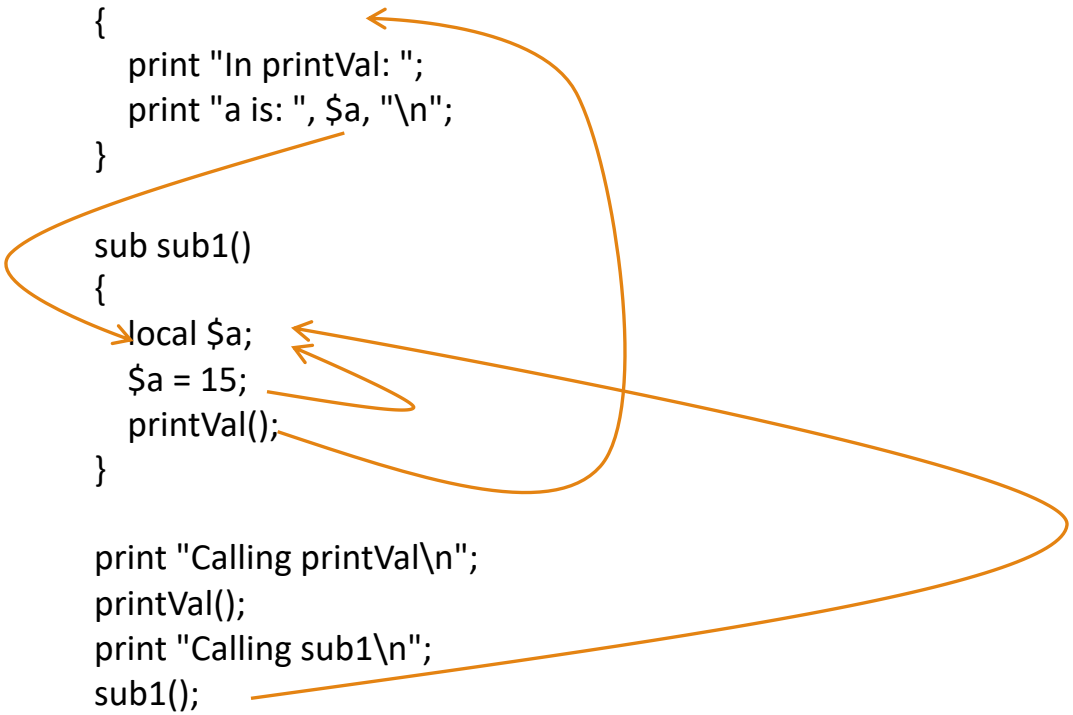
Dynamic Scope - Example

```
$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();
```



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();
```



Elaboration

Occurs at run-time

Creation of a binding

Also, allocation of space for stack variable

Also, possibly, initialization

Questions