Concepts of Programming Languages

Lecture 07 - Names

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Assignments:

Assignment #2 : due 02.19

Reading:

(Skip – Chapter 4)
Chapter 5

The first step toward wisdom is calling things by their right names.

Anonymous Chinese Proverb

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Scope
- Lifetime

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Scope
- Lifetime

Names

Not all variables have them (e.g., Perl)

Variables are characterized by attributes:

- To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

Design issues for names:

- Are names case sensitive?
- Are special words reserved words or keywords?

Name Length

If too short, they cannot be connotative

Language examples:

- FORTRAN 95: maximum of 31
- C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
- C#, Ada, and Java: no limit, and all are significant
- C++: no limit, but implementers often impose one

Special Names

PHP: all variable names must begin with dollar signs

Perl: all variable names begin with special characters, which specify the variable's type

Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Case Sensitivity

Disadvantage: readability (names that look alike are different)

- Names in the C-based languages are case sensitive
- Names in others are not: e.g., COBOL, Fortran, Basic, Pascal
- Worse in C++, Java, and C# because predefined names are mixed case (e.g. IndexOutOfBoundsException)

Special Words

Definition

A ***keyword*** is a word that is special only in certain contexts.

An aid to readability; used to delimit or separate statement clauses

A reserved word is a special word that cannot be used as a user-defined name

Usually identify major constructs: `if while switch`
or predefined identifiers: e.g., library routines

Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Scope
- Lifetime

Address

Definition

Address is the memory address with which it is associated

Each variable is associated with a memory address

A variable may have different addresses at different times during execution

A variable may have different addresses at different places in a program

If two variable names can be used to access the same memory location, they are called aliases

Aliases are created via pointers, reference variables, C and C++ unions

Binding

Definition

The term ***binding*** is an association between an entity (such as a variable) and a property (such as its value).

Name bindings play a fundamental role.

The lifetime of a variable name refers to the time interval during which memory is allocated.

Possible Binding Times

Language design time – bind operator symbols to operations

Language implementation time – bind floating point type to a representation

Compile time – bind a variable to a type in C or Java

Load time – bind a C or C++ static variable to a memory cell)

Runtime – bind a nonstatic local variable to a memory cell

Binding Time

Definition

Binding time is the time at which a binding takes place.

Definition

A binding is ***static*** if the association occurs before run-time and remains unchanged throughout program execution.

Definition

A binding is ***dynamic*** if the association occurs at run-time or can change during execution of the program.

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Scope
- Lifetime

Value

Definition

Value is the contents of the location with which the variable is associated

L-value - use of a variable name to denote its address.

Ex: x = . . .

R-value - use of a variable name to denote its value.

Ex: . . . = . . . x . . .

Some languages support/require explicit dereferencing.

Ex: x := !y + 1

Pointer Example

The unary star $\star$ deferences a pointer variable.

Pointer Example

```
int x, y;
int *p;
x = *p;
*p = y;
```

What is happening in this statement?

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type - we will cover in separate lecture
- Scope
- Lifetime

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Scope
- Lifetime

Scope

Definition

The ***scope*** of a name is the collection of statements which can access the name binding.

Definition

In ***static scoping***, a name is bound to a collection of statements according to its position in the source program.

Most modern languages use static (or lexical) scoping.

Two different scopes are either ***nested*** or ***disjoint***.

In disjoint scopes, same name can be bound to different entities without interference.

Scope

What constitutes a scope?

	Algol	C	Java	Ada
Package	n/a	n/a	yes	yes
Class	n/a	n/a	nested	yes
Function	nested	yes	yes	nested
Block	nested	nested	nested	nested
For Loop	no	no	yes	automatic

Scope

Definition

The scope in which a name is defined or delared is called its ***defining scope***

Definition

A reference to a name is ***nonlocal*** if it occurs in a nested scope of the defining scope; otherwise, it is ***local***

Example Scope in C

```c
void sort (float a[ ], int size) {
   int i, j;
   for (i = 0; i < size; i++)
      for (j = i + 1; j < size; j++)
         if (a[j] < a[i]) {
            float t;
            t = a[i];
            a[i] = a[j];
            a[j] = t;
         }
}
```

Example Scope in C

```
1    for (int i = 0; i < 10; i++) {
2      System.out.println(i);
3      ...
10   }

...

20   System.out.println(i);
```

Example Scope in C

```
for (int i = 0; i < 10; i++) {
   System.out.println(i);
   ...
}

...

System.out.println(i);
```

Invalid Reference to i!

Scope

Definition

A ***symbol table*** is a data structure kept by a translator that allows it to keep track of each declared name and its binding.

Assume for now that each name is unique within its local scope.

The data structure can be any implementation of a dictionary, where the name is the key.

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
   (a) If found, return the binding.

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
   (a) If found, return the binding.
   (b) Otherwise, repeat the process on the next dictionary down in the stack.

Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
   (a) If found, return the binding.
   (b) Otherwise, repeat the process on the next dictionary down in the stack.
   (c) If the name is not found in any dictionary, report an error.

Example Scope in C

```c
void sort (float a[ ], int size) {
   int i, j;
   for (i = 0; i < size; i++)
      for (j = i + 1; j < size; j++)
         if (a[j] < a[i]) {
            float t;
            t = a[i];
            a[i] = a[j];
            a[j] = t;
         }
}
```

Example Scope in C

```
1  | void sort (float a[ ], int size) {
2  |    int i, j;
3  |    for (i = 0; i < size; i++)
4  |       for (j = i + 1; j < size; j++)
5  |          if (a[j] < a[i]) {
6  |             float t;
7  |             t = a[i];
8  |             a[i] = a[j];
9  |             a[j] = t;
10 |          }
11 | }
```

Stack of dictionaries at line 7:

```
<t, 6>
<j, 4> <i, 3> <size,1> <a, 1>
<sort, 1>
```

Example Scope in C

```
 1 void sort (float a[ ], int size) {
 2    int i, j;
 3    for (i = 0; i < size; i++)
 4       for (j = i + 1; j < size; j++)
 5          if (a[j] < a[i]) {
 6             float t;
 7             t = a[i];
 8             a[i] = a[j];
 9             a[j] = t;
10          }
11 }
```

Stack of dictionaries at line 4 and 11:

```
<j, 4> <i, 3> <size,1> <a, 1>
<sort, 1>
```

Resolving References

Definition

For static scoping, the ***referencing environment*** for a name is its defining scope and all nested subscopes.

The referencing environment defines the set of statements which can validly reference a name.

Resolving References

```
int h, i;
void B(int w) {
  int j, k;
  i = 2 * w;
  w = w + 1;
  ...
}
void A (int x, int y) {
  float i, j;
  B(h);
  i = 3;
  ...
}
void main() {
  int a, b;
  h = 5; a = 3; b = 2;
  A(a, b);
  B(h);
  ...
}
```

Resolving References

```
int h, i;
```

1. Outer scope: `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
2. Function B: `<w, 2> <j, 3> <k, 3>`
3. Function A: `<x, 8> <y, 8> <i, 9> <j, 9>`
4. Function main: `<a, 15> <b, 15>`

Resolving References

```
2 | void B(int w) {
3 |   int j, k;
4 |   i = 2 * w;
5 |   w = w + 1;
6 |   ...
7 | }
```

1. Outer scope: `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
2. Function B: `<w, 2> <j, 3> <k, 3>`
3. Function A: `<x, 8> <y, 8> <i, 9> <j, 9>`
4. Function main: `<a, 15> <b, 15>`

Resolving References

```
8  | void A (int x, int y) {
9  |   float i, j;
10 |   B(h);
11 |   i = 3;
12 |   ...
13 | }
```

1. Outer scope: `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
2. Function B: `<w, 2> <j, 3> <k, 3>`
3. Function A: `<x, 8> <y, 8> <i, 9> <j, 9>`
4. Function main: `<a, 15> <b, 15>`

Resolving References

```
14 | void main() {
15 |   int a, b;
16 |   h = 5; a = 3; b = 2;
17 |   A(a, b);
18 |   B(h);
19 |   ...
20 | }
```

1. Outer scope: `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`
2. Function B: `<w, 2> <j, 3> <k, 3>`
3. Function A: `<x, 8> <y, 8> <i, 9> <j, 9>`
4. Function main: `<a, 15> <b, 15>`

Resolving References

```
2 void B(int w) {
3   int j, k;
4   i = 2 * w;
5   w = w + 1;
6   ...
7 }
```

Symbol Table Stack for Function B:

```
<w, 2> <j, 3> <k, 3>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Resolving References

```
8  | void A (int x, int y) {
9  |   float i, j;
10 |   B(h);
11 |   i = 3;
12 |   ...
13 | }
```

Symbol Table Stack for Function A:

```
<x, 8> <y, 8> <i, 9> <j, 9>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Resolving References

```
14 | void main() {
15 |   int a, b;
16 |   h = 5; a = 3; b = 2;
17 |   A(a, b);
18 |   B(h);
19 |   ...
20 | }
```

Symbol Table Stack for Function main:

```
<a, 15> <b, 15>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
```

Resolving References

```
 1 | int h, i;
 2 | void B(int w) {
 3 |   int j, k;
 4 |   i = 2 * w;
 5 |   w = w + 1;
 6 |   ...
 7 | }
 8 | void A (int x, int y) {
 9 |   float i, j;
10 |   B(h);
11 |   i = 3;
12 |   ...
13 | }
```

```
14 | void main() {
15 |   int a, b;
16 |   h = 5; a = 3; b = 2;
17 |   A(a, b);
18 |   B(h);
19 |   ...
20 | }
```

Line	Ref	Decl
4	i	1
10	h	1
11	i	9
16	h	1
18	h	1

Dynamic Scoping

Definition

In ***dynamic scoping***, a name is bound to its most recent declaration based on the program's call history.

Used by early Lisp, APL, Snobol, Perl.

Symbol table for each scope built at compile time, but managed at run time.

Scope pushed/popped on stack when entered/exited.

Dynamic Scoping

```
int h, i;
void B(int w) {
  int j, k;
  i = 2 * w;
  w = w + 1;
  ...
}
void A (int x, int y) {
  float i, j;
  B(h);
  i = 3;
  ...
}
void main() {
  int a, b;
  h = 5; a = 3; b = 2;
  A(a, b);
  B(h);
  ...
}
```

Dynamic Scoping

Call history :

`main (17)` $\rightarrow$ `A (10)` $\rightarrow$ `B`

Function	Dictionary
B	`<w, 2> <j, 3> <k, 3>`
A	`<x, 8> <y, 8> <i, 9> <j, 9>`
main	`<a, 15> <b, 15>` `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`

Reference to `i (4)` resolves to `<i, 9>` in `A`.

Dynamic Scoping

```
int h, i;
void B(int w) {
  int j, k;
  i = 2 * w;
  w = w + 1;
  ...
}
void A (int x, int y) {
  float i, j;
  B(h);
  i = 3;
  ...
}
```

```
void main() {
  int a, b;
  h = 5; a = 3; b = 2;
  A(a, b);
  B(h);
  ...
}
```

Dynamic Scoping

Call history :

$$\texttt{main (18)} \rightarrow \texttt{B}$$

Function	Dictionary
B	`<w, 2> <j, 3> <k, 3>`
main	`<a, 15> <b, 15>` `<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>`

Reference to `i (4)` resolves to `<i, 1>` in global scope.

Visibility

Definition

A name is ***visible*** if its referencing environment includes the reference and the name is not redeclared in an inner scope.

Definition

A name redeclared in an inner scope effectively ***hides*** the outer declaration.

Some languages provide a mechanism for referencing a hidden name; e.g.: `this.x` in C++/Java.

Visibility

```
public class Student {
  private String name;
  public Student (String name, ...)  {
    this.name = name;
    ...
  }
}
```

Ada Program

```
procedure Main is
  x :  Integer;
  procedure p1 is
    x :  Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

Ada Program

```
procedure Main is
  x :  Integer;
  procedure p1 is
    x :  Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

`x` in `p2`?

Ada Program

```
procedure Main is
  x :  Integer;
  procedure p1 is
    x :  Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

`x` in `p2`?

Ada Program

```
procedure Main is
  x : Integer;
  procedure p1 is
    x : Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

`x` in `p2`?

`x` in `p1`?

Ada Program

```
procedure Main is
  x :  Integer;
  procedure p1 is
    x :  Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

`x` in `p2`?

`x` in `p1`?

Ada Program

```
procedure Main is
  x :  Integer;
  procedure p1 is
    x :  Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

`x` in `p2`?

`x` in `p1`?

`x` in `p3`?

Ada Program

```
procedure Main is
  x :  Integer;
  procedure p1 is
    x :  Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

`x` in `p2`?

`x` in `p1`?

`x` in `p3`?

Ada Program

```
procedure Main is
  x : Integer;
  procedure p1 is
    x : Float;
    procedure p2 is
    begin
      ...x ...
    end p2;
  begin
    ...x ...
  end p1;
  procedure p3 is
  begin
    ...x ...
  end p3;
begin
  ...x ...
end main;
```

x in p2?

x in p1?

x in p3?

x in Main?

Overloading

Definition

Overloading uses the number or type of parameters to distinguish among identical function names or operators.

Examples:

- +, -, *, / can be float or int
- + can be float or int addition or string concatenation in Java
- `System.out.print(x)` in Java

Overloading

Modula: library functions

- `Read( )` for characters
- `ReadReal( )` for floating point
- `ReadInt( )` for integers
- `ReadString( )` for strings

Overloading

```
public class PrintStream extends FilterOutputStream {
  ...
  public void print(boolean b);
  public void print(char c);
  public void print(int i);
  public void print(long l);
  public void print(float f);
  public void print(double d);
  public void print(char[ ] s);
  public void print(String s);
  public void print(Object obj);
}
```

Variables

Variables can be characterized as a sextuple of attributes:

- Name
- Address
- Value
- Type
- Scope
- Lifetime

Lifetime

Definition

The ***lifetime*** of a variable is the time interval during which the variable has been allocated a block of memory.

Earliest languages used static allocation.

Algol introduced the notion that memory should be allocated/deallocated at scope entry/exit.

Remainder of section considers mechanisms which break scope equals lifetime rule.

Lifetime

C:

- Global compilation scope: static
- Explicitly declaring a variable static

Java also allows a variable to be declared static