22c:111 Programming Language Concepts

Fall 2008

Names
4.1 Syntactic Issues
4.2 Variables
4.3 Scope
4.4 Symbol Table
4.5 Resolving References
4.6 Dynamic Scoping
4.7 Visibility
4.8 Overloading
4.9 Lifetime
Recall that the term *binding* is an association between an entity (such as a variable) and a property (such as its value).

A binding is *static* if the association occurs before run-time.

A binding is *dynamic* if the association occurs at run-time.

Name bindings play a fundamental role.

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

Lexical rules for names.
Collection of reserved words or keywords.
Case sensitivity
   C-like: yes
   Early languages: no
   PHP: partly yes, partly no
Reserved Words

Cannot be used as *Identifiers*

Usually identify major constructs: *if while switch*

Predefined identifiers: e.g., library routines
Variables

Basic bindings

- Name
- Address
- Type
- Value
- Lifetime
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:

int x, y;

int *p;

x = *p;

*p = y;
Scope

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

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.

What constitutes a scope?
<table>
<thead>
<tr>
<th></th>
<th>Algol</th>
<th>C</th>
<th>Java</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>n/a</td>
<td>n/a</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Class</td>
<td>n/a</td>
<td>n/a</td>
<td>nested</td>
<td>yes</td>
</tr>
<tr>
<td>Function</td>
<td>nested</td>
<td>yes</td>
<td>yes</td>
<td>nested</td>
</tr>
<tr>
<td>Block</td>
<td>nested</td>
<td>nested</td>
<td>nested</td>
<td>nested</td>
</tr>
<tr>
<td>For Loop</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>automatic</td>
</tr>
</tbody>
</table>
The scope in which a name is defined or declared is called its *defining scope*.

A reference to a name is *nonlocal* if it occurs in a nested scope of the defining scope; otherwise, it is *local*.
void sort (float a[ ], int size) {
    int i, j;
    i = j = 0;
    for (i = 0; i < size; i++)
        for (j = i + 1; j < size; j++)
            if (a[j] < a[i]) {
                float t1,t2;
                t1 = a[i];
                a[i] = a[j];
                a[j] = t2;
            }
    j = i + 1
    i = 2*j }

```c
void sort (float a[], int size) {  // C language
    int i, j;
    i = j = 0;
    for (i = 0; i < size; i++)  // i, size local
        for (j = i + 1; j < size; j++)  // j local
            if (a[j] < a[i]) {  // a, i, j local
                float t1, t2;
                t1 = a[i];    // t1 local; a, i nonlocal
                a[i] = a[j];    // j nonlocal
                a[j] = t2;     // t2 local
            }
    j = i + 1;  // i, j local
};
    i = 2*j }  // i, j local
```
void sort (float a[], int size) { // C++ language
    int i, j;
    i = j = 0;
    for (int i = 0; i < size; i++) { // i local, size nonlocal
        for (int j = i + 1; j < size; j++)  // j local
            if (a[j] < a[i]) {  // j local,  a, i nonlocal
                float t1, t2;
                t1 = a[i];     // t1 local; a, i nonlocal
                a[i] = a[j];    // j nonlocal
                a[j] = t2;     // t2 local
            }

        j = i + 1;       // i local, j nonlocal
    }

    i = 2*j }        // i, j local
for (int i = 0; i < 10; i++) {
    System.out.println(i);
    ...
    ...
} // invalid reference to i
Symbol Table

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.
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.
void sort (float a[ ], int size) {  // C++ language
int i, j;
i = j = 0;
for (int i = 0; i < size; i++) { // i local, size nonlocal
    for (int j = i + 1; j < size; j++)  // j local
        if (a[j] < a[i]) {  // j local, a, i nonlocal
            float t1,t2;
t1 = a[i];       // t1 local; a, i nonlocal
            a[i] = a[j];    // j nonlocal
            a[j] = t2;     // t2 local
        }
    j = i + 1;
};
j = i + 1;        // i local, j nonlocal
};
i = 2*j }        // i, j local
Example: previous C++ program

Bindings: \( \langle \text{var name, line location in code} \rangle \)

Dictionary Stack at line 7:

\[
\begin{align*}
\{ & \langle t_1, 7 \rangle, \langle t_2, 7 \rangle \} \\
& \{ \langle j, 4 \rangle \} \\
& \{ \langle i, 4 \rangle \} \\
& \{ \langle a, 1 \rangle, \langle \text{size}, 1 \rangle, \langle i, 2 \rangle, \langle j, 2 \rangle \} 
\end{align*}
\]
Resolving References

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.
```c
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 }
```
1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
2. Function B: <w, 2> <j, 3> <k, 4>
3. Function A: <x, 8> <y, 8> <i, 9> <j, 9>
4. Function main: <a, 15> <b, 15>
Symbol Table Stack for Function B:
  <w, 2> <j, 3> <k, 4>
  <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
Symbol Table Stack for Function A:
  <x, 8> <y, 8> <i, 9> <j, 9>
  <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
Symbol Table Stack for Function main:
  <a, 15> <b, 15>
  <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
<table>
<thead>
<tr>
<th>Line</th>
<th>Reference</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>h</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>i</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>h</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>h</td>
<td>1</td>
</tr>
</tbody>
</table>
Dynamic Scoping

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

Used be 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.
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 }
Using Figure 4.2 as an example: call history

main (17) → A (10) → B

<table>
<thead>
<tr>
<th>Function</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>&lt;w, 2&gt; &lt;j, 3&gt; &lt;k, 3&gt;</td>
</tr>
<tr>
<td>A</td>
<td>&lt;x, 8&gt; &lt;y, 8&gt; &lt;i, 9&gt; &lt;j, 9&gt;</td>
</tr>
<tr>
<td>main</td>
<td>&lt;a, 15&gt; &lt;b, 15&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;h, 1&gt; &lt;i, 1&gt; &lt;B, 2&gt; &lt;A, 8&gt; &lt;main, 14&gt;</td>
</tr>
</tbody>
</table>

Reference to i (4) resolves to <i, 9> in A.
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 }
Using Figure 4.2 as an example: call history

main (17) → B

<table>
<thead>
<tr>
<th>Function</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>&lt;w, 2&gt; &lt;j, 3&gt; &lt;k, 3&gt;</td>
</tr>
<tr>
<td>main</td>
<td>&lt;a, 15&gt; &lt;b, 15&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;h, 1&gt; &lt;i, 1&gt; &lt;B, 2&gt; &lt;A, 8&gt; &lt;main, 14&gt;</td>
</tr>
</tbody>
</table>

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

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

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.
public class Student {
    private String name;
    public Student (String name, ...) {
        this.name = name;
        ...
    }
    }
}
procedure Main is
  x : Integer;
procedure p1 is
  x : Float;
procedure p2 is
begin
  ... x ...
end p2;
begin
  ... x ...
end p1;
begin
  ... x ...
end p3;

begin
  ... x ...
end Main; -- Ada
-- x in p2?
-- x in p1? Main.x?
-- x in p3? p1.x?
-- x in Main?
Overloading

*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
Modula: library functions

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

Lifetime

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.
C:

- Global compilation scope: static
- Explicitly declaring a variable `static`
- Remark: Java also allows a variable to be declared `static`