NJIT

New Jersey's Science & Technology University

THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Names

Names

- What do they look like in the language?
- What do they mean?
- Important Terms:
  - Binding – what’s the name associated with?
  - Scope – where’s the name visible?
  - Lifetime – how long does the name last?

Bindings

- A *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 in programming languages.

Syntactic Issues

- What are the lexical rules for names?
- Is there a collection of reserved words or keywords?
- Are names case sensitive?
  - C-like: yes
  - Early languages: no
  - SQL: no
  - PHP: partly yes, partly no

Reserved Words

- Cannot be used as *Identifiers*
- Usually identify major constructs: *if while switch*
- Predefined identifiers
  - In some languages, library routines might be reserved words
  - In other languages, library routines are just names and are not reserved words

Variables

Basic bindings

- Name
- Address
- Type
- Value
- Lifetime

Variables

Basic bindings

- Name – follows the lexical rules of the language
- Address – where in memory??
- Type – *more on this later*
- Value – what is stored in the memory that is bound to the name
- Lifetime

Addresses

- Memory for a running program is divided into several regions or segments
  - The operating system may control permissions on memory
  - The runtime is responsible for managing memory
- Possible Regions
  - Code
  - Data
  - Heap
  - Stack

What Memory Is Used?

```
int outside; // external variable
    // put in a data segment by compile/link

// the code for the function is in a code segment
void myFunction()
{
  // these two variables are on the stack
```

L-value - use of a variable in the case where you care about it’s address

Ex: x = …

R-value - use of a variable in the case where you care about the value stored in the variable

Ex: … = … x …

```
// These are integers
int x,y,z;     // rvalue is the integer
           // lvalue is the location of the integer

// Pointer:
int *p;   // rvalue is the pointer
           // lvalue is where the pointer is stored
           // *p used as an rvalue:
                // rvalue of what the pointer points to
           // *p used as an lvalue:
                // lvalue of what the pointer points to

p = &x;     /* store x's lvalue in p */
y = *p;     /* store what p points to (*p's rvalue) in y */
*p = z;     /* store z in what p points to (*p's lvalue) */
```

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?
  - Global
    - In Java everything has to be in a class; there’s no global functions or global data
    - C and C++ have global functions and data
  - File (compilation unit)
  - Certain subdivisions of the code

Possible Scopes

- Java Packages
- C++ Namespaces
- Classes (which may be nested in other classes)
- Functions
- Blocks
- For loops

- The scope in which a name is defined or delared 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) {

  for (int i = 0; i < size; i++)  // i local; size not
    for (int j = i + 1; j < size; j++) // j local; i,size not
       if (a[j] < a[i]) {  // j local; a, i, nonlocal
           float t;
           t = a[i];     // t local; a, i nonlocal
           a[i] = a[j];  // a, i, j nonlocal
           a[j] = t;     // t local; a, j nonlocal
      }
}
```

```
for (int i = 0; i < 10; i++) {
    cout << i << endl;
}

if(i == 10) {
  // this is an invalid reference
  // to the i from the loop:
  // this reference is out of scope
}
```

Controlling The Scope

- Some languages give the programmer more control over scope
- static
  - Variables, classes, class members
- Class members (C++ and Java)
  - public
  - private
  - protected
- Scope resolution op (::) in C++

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.
- 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) {

  for (int i = 0; i < size; i++)  // I local, size not
    for (int j = i + 1; j < size; j++)
       if (a[j] < a[i]) {  // j local; a, i, nonlocal
           float t;
           t = a[i];     // t local; a, i nonlocal
           a[i] = a[j];  // a, I, j nonlocal
           a[j] = t;     // t local; a, j nonlocal
      }
}
```

C sort program, stack of dictionaries

At line 4:

<i, 3>

<sort, 1> <size,1> <a, 1>

At line 5:

<j, 4>

<i, 3>

<sort, 1> <size,1> <a, 1>

At line 7:

<t, 6>

<j, 4>

<i, 3>

<sort, 1> <size,1> <a, 1>

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.

```
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);
    ...
}
```

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>

Symbol Table Stack for Function B:

    <w, 2> <j, 3> <k, 3>

    <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>

Line	Reference	Declaration
4	i	1
10	h	1
11	i	9
16	h	1
18	h	1

Dynamic Scoping

- In dynamic scoping, a name is bound to its most recent declaration based on the program’s call history.
- Used in early Lisp, APL, Snobol, Perl.
- Symbol table for each scope built at compile time, but managed at run time.

```
function big() {
    function sub1() {
        var x = 7;
        sub2();
    }

    function sub2() {
        var y = x;        // which x??
    }

    var x = 3;
    sub1();
}
```

Dynamic Scoping

call history

big() □sub1() □sub2()

Reference to x on line 7 resolves to the x on line 2

If instead of the call to sub1(), the sequence was:

big() □sub2()

Then the reference to x on line 7 would resolve to the x on line 9

Visibility

- A name is *visible* if its referencing environment includes the reference and the name is not redeclared in an inner scope.
- A name that is redeclared in an inner scope effectively *hides* the outer declaration.
- Some languages provide a set of mechanisms for referencing a limited set of hidden names
  - this.x in Java
  - this->x in C++
  - C++ scope resolution operator ::

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

```
1 int count;
2 int function() {
3     int count;
4     count = 0; // the local count on line 3
5     ::count = 1;    // the global count on line 1
6 }
```

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

```
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.

Static Variables

- C
  - Global compilation scope: variables exist forever
  - Explicitly declaring a variable static
    - Variables exist forever
    - Visibility is changed – Only visible when in scope
  - Static functions: only visible in the file they're defined in
- C++
  - Static classes, class members, methods
- Java also has static variables

NJIT

THE EDGE IN KNOWLEDGE