Visibility and Symbol Tables

CS152

Chris Pollett

Oct. 6, 2008.

Outline

- Visibility
- Symbol Table

Introduction

- Recall last Wednesday we were talking about bindings.
- A binding associates an attribute to name in programming language.
- One common source of bindings is declarations and these commonly occur with blocks.
- The scope of a binding is the region of the program over which the binding is maintained.
- For example, C has the **declaration before use** rule. So the scope of a binding extends from the point just after the declaration to the end of the block in which it is located.

Visibility

- Consider:

```
int x;
void p() {
   char x;
    x='a' //assigns to char x
}
void main(){
    x=2; //assigns to global x
}
```

- Declaration in nested blocks (in this case in p) take precedence over previous declarations.
- The scope of the global x is the whole program but there is a so-called **scope hole** over function p.
- The **visibility** of a declaration includes only those regions where the bindings of a declaration apply.

More on Visibility

- In C++, one could use the scope resolution operator to access the global x within function p. I.e., we could write ::x to access it in p.
- In Ada, you can give names to scopes with a syntax like:
  name : declare … begin … end .
- You can access variable declared in a given scope using a syntax like name.varname ; this is called **visibility by selection**.
- In C, global variable declarations can be accessed across files using the **extern**.
- The declaration extern int a; //indicates program will be linked to another compiled file where a had global scope.

The Symbol Table

- Last week we said that information about static bindings are maintained by the compiler in a symbol table.
- This table might be implemented using a data structure for tables as one would see in a class like CS146. For example, hash tables.
- It supports such operations as insert, lookup, and delete on names.
- To maintain this table in a lexically scoped language requires that declarations be processed in a stacklike fashion: On entry to a block, all declarations of that block are processed and the corresponding bindings added to the symbol table; on exit from the block, these bindings are "popped" restoring the previous bindings that may have existed.

Example Program to Illustrate Symbol Table.

```
int x;
char y; // (*)
void p() {
   double x;
   … // (**)
   { int y[10]; // (***)
   }
} // (****)
void q() { int y; /* (#) */} // (##)
int main() {char x; /* (###) */}
```

Symbol Table Example

- Names in the above program are x,y, p, q, main.
- x, y are associated with three different declarations and scopes.
- At (**), the symbol table might look like:
  - (x, (double local to p), (int global))
  - (y, (char global))
  - (p, (void function))
- At (***), the symbol table might look like:
  - (x, (double local to p), (int global))
  - (y, (int array local to nested block in p), (char global))
  - (p, (void function))

More Symbol Table Example

- When the inner block of p is finished being processed the declaration of y would be popped, when p is finished being processed the local declaration of x would also be popped leaving the symbol table as:
  - (x, (int global))
  - (y, (char global))
  - (p, (void function))
- As q begins to be processed at (#), the symbol table would look like:
  - (x, (int global))
  - (y, (int local to q), (char global))
  - (p, (void function))
  - (q, (void function))
- You should try to work out what the table looks like at (##) and (###).

Static Versus Dynamic Scoping

- The above symbol tables illustrate the kind of scoping that might occur in a compiler as it parses a program prior to execution.
- This is called **static scoping**.
- If the symbol table is managed during execution, then declarations are processed as they are encountered along an execution path through program. This is called **dynamic scoping**.

Example Program to Illustrate Dynamic Scoping

```
#include <stdio.h>
int x = 1;
char y = 'a';
void p() { double x = 2.5;
    printf("%c\n", y); // (***)
    {
        int y[10];
    }
}
void q() { int y = 42;
    printf("%d\n", x); //(**)
    p();
}
int main() { char x= 'b'; // (*)
   q();
   return 0;
}
```

More Dynamic Scoping Example

- If the symbol table is constructed dynamically, then it would be constructed beginning with the execution of main.
- Global declaration that occur before main would have been processed, and the symbol table at (*) might look like:

  (x, (char = 'b' local to main), (int = '1' global))

  (y, (char = 'a' global))

  (p, (void function))

  (q, (void function))

  (main, (int function))

Yet More Dynamic Scoping Example

- After main calls q, we begin processing q and the symbol table at (**) would look like:
  - (x, (char = 'b' local to main), (int = '1' global))
  - (y, (int = 42 local to q), (char = 'a' global))
  - (p, (void function))
  - (q, (void function))
  - (main, (int function))
- Notice this is quite different from the symbol on entry to q when we were doing static processing.
- Notice also each call to q might have a different symbol table on entry to q depending on the execution path.
- Try to figure out what the table would look like at (***).
- What is the output of the program using static scoping versus dynamic scoping?