Symbol Table Implementations

- Symbol table will be used to answer two questions:
  1. Given a declaration of a name, is there already a declaration of the same name in the current scope
     - i.e., is it multiply declared?
  2. Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?

Note

- Generally, symbol table is only needed to answer those two questions, i.e.,
  - once all declarations have been processed to build the symbol table,
  - and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry,
  - then the symbol table itself is no longer needed
    - because no more lookups based on name will be performed

Assumptions

- For this work, assume we:
  - use static scoping
  - require that *all* names be declared before they are used
  - do not allow multiple declarations of a name in the same scope
    - even for different kinds of names
  - *do* allow the same name to be declared in multiple nested scopes
    - but only once per scope
  - use the same scope for a method's parameters and for the local variables declared at the beginning of the method

What operations do we need?

- Given the above assumptions, we will need:
  1. Look up a name in the current scope only
     - to check if it is multiply declared
  2. Look up a name in the current and enclosing scopes
     - to check for a use of an undeclared name, and
     - to link a use with the corresponding symbol-table entry
  3. Insert a new name into the symbol table with its attributes
  4. Do what must be done when a new scope is entered
  5. Do what must be done when a scope is exited

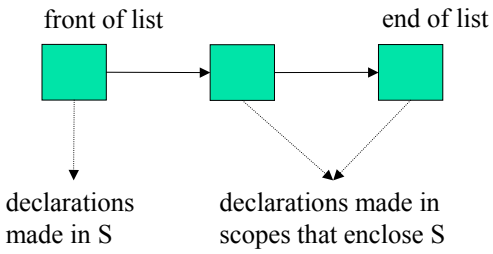
Some possible symbol table implementations

1. a list of tables
2. a table of lists

- For each approach, we will consider
  - what must be done when entering and exiting a scope,
  - when processing a declaration, and
  - when processing a use
- Simplification:
  - assume each symbol-table entry includes only:
    - the symbol name
    - its type
    - the nesting level of its declaration

Method 1: List of Hashtables

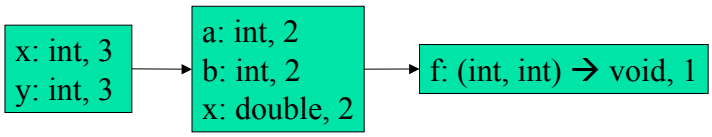
- The idea:
  - symbol table = a list of hashtables,
  - one hashtable for each currently visible scope.
- When processing a scope S:



Example:

```
void f(int a, int b) {
  double x;
  while (...) { int x, y; ... }
}
void g() { f(); }
```

- After processing declarations inside the while loop:



List of Hash Tables - Operations

- On scope entry:
  - increment the current level number and add a new empty hashtable to the front of the list.
- To process a declaration of x:
  - look up x in the first table in the list
    - If it is there, then issue a "multiply declared variable" error;
    - otherwise, add x to the first table in the list

List of Hash Tables - Operations

- To process a use of x:
  - look up x starting in the first table in the list;
    - if it is not there, then look up x in each successive table in the list
    - if it is not in *any* table then issue an "undeclared variable" error
- On scope exit,
  - remove the first table from the list and decrement the current level number

Inserting Method/Function Names

- Method names belong in the hashtable for the outermost scope
  - Not in the same table as the method's variables
- For example, in the previous example:
  - Method name f is in the symbol table for the outermost scope
  - Name f is *not* in the same scope as parameters a and b, and variable x
  - This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table

Running times for each operation:

1. **Scope entry**:
   - time to initialize a new, empty hashtable;
   - probably proportional to the size of the hashtable
2. **Process a declaration:**
   - using hashing, constant expected time (O(1))
3. **Process a use**:
   - using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined
4. **Scope exit**:
   - time to remove a table from the list, which should be O(1) if garbage collection is ignored

Scoping Example

- C++ does not use exactly the scoping rules that we have been assuming
  - In particular, C++ **does** allow a function to have both a parameter and a local variable with the same name
    - any uses of the name refer to the local variable
  - Consider the following code. What is the symbol table as it would be after processing the declarations in the body of *f* under:
    - the scoping rules we have been assuming
    - C++ scoping rules

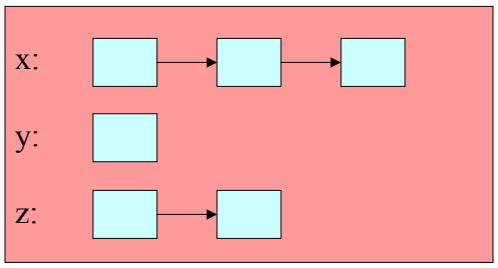
```
void g(int x, int a) { }
void f(int x, int y, int z) {
  int a, b, x; ...
}
```

Scoping Example (cont)

- Questions:
  - Which of the four operations described above
    - scope entry,
    - process a declaration,
    - process a use,
    - scope exit
  - would we need to change to use the following rules for name reuse instead of C++ rules:
    - the same name can be used within one scope as long as the uses are for different kinds of names, and
    - the same name *cannot* be used for more than one variable declaration in a nested scope

Method 2: Hash Table of Lists

- the idea:
  - when processing a scope S, the structure of the symbol table is:



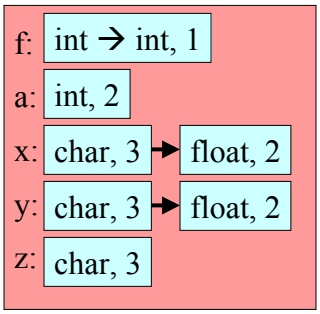
Definition

- There is just one big hashtable, containing an entry for each variable for which there is
  - some declaration in scope S or
  - in a scope that encloses S
- Associated with each variable is a list of symbol-table entries
  - The first list item corresponds to the most closely enclosing declaration;
  - the other list items correspond to declarations in enclosing scopes

Example

```
int f (int a) {
  float x, y;
  while (...) {
    char x, y, z;
  }
}
void g () {
  int x;
  f(1);
}
```

- After processing the declarations inside the while loop:



Nesting level information is crucial

- the level-number attribute stored in each list item enables us to determine whether the most closely enclosing declaration was made
  - in the current scope or
  - in an enclosing scope

Hash Table of Lists: Operations

- On scope entry:
  - increment the current level number
- To process a declaration of x:
  - look up x in the symbol table
    - If x is there, fetch the level number from the first list item.
      - If that level number = the current level then issue a "multiply declared variable" error;
      - otherwise, add a new item to the front of the list with the appropriate type and the current level number

Hash Table of Lists: Operations

- To process a use of x:
  - look up x in the symbol table
  - If it is not there, then issue an "undeclared variable" error
- On scope exit:
  - scan all entries in the symbol table, looking at the first item on each list
  - If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry)
  - Finally, decrement the current level number

Running times

1. **Scope entry**:
   - time to increment the level number, O(1)
2. **Process a declaration**:
   - using hashing, constant expected time O(1)
3. **Process a use**:
   - using hashing, constant expected time O(1)
4. **Scope exit**:
   - time proportional to the number of names in the symbol table (or perhaps even the size of the hashtable if no auxiliary information is maintained to allow iteration through the non-empty hashtable buckets)

Symbol Table Example

- Assume symbol table is implemented using a hash table of lists
- How does symbol table change in processing the following?

```
void g(int x, int a) {
    double d;
    while (...) {
        int d, w;
        double x, b;
        if (...) { int a,b,c; }
    }
    while (...) { int x,y,z; }
}
```