COEN 175

Lecture 7: Symbol Tables

Uniqueness Checks

- At the time of its declaration, information about an identifier must be recorded for later use.
- There should be one and only one declaration.
- The central repository for all such information is called the **symbol table**.
- A **symbol** is simply any named object we store information about:
  - A variable such as `x`
  - A function such as `main`
  - A named type such as `uint8_t`

Basic Table Operations

- What kinds of operations do we need on our table?
  - `insert(symbol)`: add a symbol to the table
  - `find(name)`: return a symbol given its name
- At declaration time, we do an `insert` operation, which will fail if the name is already in use.
- Upon seeing a use, we do a `find` operation, which will return the associated symbol for our use.
- Unfortunately, most languages have **scoping** that complicates our design.

Scoping

- A **scope** is simply a region of code over which a given name is valid. You can think of it as the lifetime of the name.
- A name is said to be **bound** to a declaration.
- Two commonly used kinds of scoping:
  - **Static scoping**: the declaration to which a name is bound is known at compile time
  - **Dynamic scoping**: the declaration to which a name is bound is not known until run time
- Most languages, including C, use static scoping.

Static Nested Scoping

- Most languages implement **static nested scoping**:
  - Scopes can be nested hierarchically within each other.
  - An inner scope must be contained completely within an outer scope. It cannot "span" two different outer or enclosing scopes.
- Most languages also allow **shadowing**:
  - A name may be redeclared within an inner scope.
  - When a name is used, it is bound to the nearest or "innermost" declaration.
  - Any object with that name declared in an outer scope will be inaccessible.

Example: Shadowing

```
int x, y;

int f(int y, int z) {          // shadowing of global y
    int a, b;

    {
        int x, y;              // shadowing of x and y
        x = a + y;             // assigns to local x
    }

    x = f(a, y);               // assigns to global x
}

int g(int f) {                 // shadowing of global f
    f = 1;                     // assigns to parameter f
}
```

Symbol Table Design

- Supporting static nested scoping requires a redesign of our symbol table.
  - Before we had one mapping from names to symbols.
  - We now need a mapping for each scope.
- How can we model the nesting of scopes?
- We can use a **stack**:
  - When a scope is created or "**opened**" then we **push** a new scope onto the stack.
  - When a scope is destroyed or "**closed**" then we **pop** it from the stack.

Symbol Table Operations

- Our new design requires a new set of operations:
  - `open(scope)`: create and push a new scope, which is linked to the given enclosing scope
  - `close()`: pop the topmost scope
  - `insert(scope, symbol)`: insert a symbol into a scope
  - `find(scope, name)`: search a given scope for a name
  - `lookup(scope, name)`: starting with a given scope, search it **and all enclosing** scopes for a name
- The key operation is `lookup`, which will find the nearest declaration.

Reporting Errors

- If a `lookup` operation fails, then the name is **undeclared**.
- If an `insert` operation fails, then the name is **previously declared**.
- Some languages may allow redeclarations:
  - In C (and Simple C), a global name may be declared multiple times so long as all such declarations have identical types.
  - We must first do a `find` operation and compare types.
  - However, a global object may be defined only once.

Example: Scoping Errors

```
int x, a[10];
int f(int a, int b);

int f(int x, int y) {
    int y;                  // redeclaration of y

    {
        int w;
        w = x + z;          // z undeclared
        y = g();            // error? it depends if implicit
    }                       // declarations are allowed

    w = a[x];               // w undeclared
}

int a[5];                   // conflicting types for a
int f(int x, int y) {}      // redefinition of f
```

Necessary Data Types

- Our symbol table requires three primary data types:
  - `Type`, which holds simple C information;
  - `Symbol`, which holds a name and its type;
  - `Scope`, which holds the symbols.
- The symbol table itself is a **stack** of scopes.
  - We explicitly link each scope to the next scope.
  - Thus, we create a linked list of scopes.
- The symbol table can be viewed as a **tree** of scopes.
  - We explicitly link each scope to the parent scope.
  - Thus, we create a hierarchy of scopes.